RC 20267 (11/09/95)
Computer Science

IBM Research Report

Flow-Sensitive Type Analysis for C++

Paul R. Carini, Harini Srinivasan

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Michael Hind

Department of Mathematics and Computer Science
State University of New York at New Paltz
New Paltz, NY 12561
and IBM T.J. Watson Research Center

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Flow-Sensitive Type Analysis for C++

Paul R. Carini*
Michael Hind†
Harini Srinivasan*

November 8, 1995

Abstract

Static determination of run-time types is a key analysis step for compile-time optimizations of object-oriented languages with dynamic dispatch of functions. Type information is fundamental for determining the virtual functions that can be invoked and enables a number of interprocedural analyses and transformations. For example, knowledge of the types of objects at virtual function call sites opens opportunities for inlining, which in turn facilitates a number of intraprocedural compiler optimizations. The results of type analysis may also aid function cloning based on the types of function parameters.

In this paper, we show how an existing flow-sensitive pointer alias analysis that uses a compact representation can be adapted to provide a type analysis algorithm which computes type information in a lazy fashion, thereby incurring minimal additional overhead. We show that use of the type information as soon as it becomes available can, in turn, improve the precision of the pointer alias analysis. We demonstrate that incorporating knowledge of the class hierarchy into our algorithm improves its precision, and show that our flow-sensitive method is more precise than flow-insensitive methods.

1 Introduction

One significant feature of C++ is the support for dynamic dispatch via virtual functions [12]. Such functions are invoked through a pointer, where the type of the object to which the pointer points determines the particular function that is invoked. Knowledge of these types at compile-time facilitates further analysis and is an important fundamental step for a number of compiler optimizations. Although the declared type of a C++ pointer variable cannot change, the type of the objects that it points to can change during program execution. We refer to the types of such objects as the run-time types of the pointer. Empirical studies [4] of the behavior of C++ programs have shown that static determination of the run-time types of objects is useful to resolve a number of indirect function calls. By ascertaining what classes (types) a pointer that is used to invoke a virtual function may point to, it is possible to limit the number of possible functions that can be invoked. This optimization can improve the precision of interprocedural analyses and transformations. For example, if it can be statically determined that the pointer invoking a virtual function can only point to an object of exactly one type, it is possible to convert this indirect function call into a direct function call. Further, it may also be possible to inline the direct function call. Empirical studies [5] indicate that C++ functions are usually small, which suggests that inlining can be a useful optimization.

The problem of determining the precise run-time types of objects in a language such as C++ has been shown to be NP-Hard for single levels of pointer indirection [22, 24], and undecidable for multiple levels of pointer indirection [27]. The work presented in [22, 24] also elucidates the interaction between pointer alias information and type information for languages such as C++ which use virtual function calls frequently. In particular, pointer alias information can affect the safety and precision of type information. Therefore, it is important to perform pointer alias analysis before type analysis in C++ programs. However, the precision of pointer alias information in such programs can be improved if type information is available, particularly

*IBM T.J. Watson Research Center, P.O. Box 704 Yorktown Heights, NY 10598, {carini, harini}@watson.ibm.com
†Department of Mathematics and Computer Science, State University of New York at New Paltz, New Paltz, NY 12561, and IBM T.J. Watson Research Center, hind@mcs.newpaltz.edu

at virtual function call sites. Earlier work has either focused on interleaving the two analyses for safety and precision [22, 24], or computed type information after a pointer alias analysis pass, at a loss of precision [25].

This work enhances the flow-sensitive pointer alias analysis algorithm of [8, 3] by computing type information within the same data flow context as the pointer alias computation. However, by using a compact representation to represent pointer alias information, the computation of type information is deferred until the type information is required. This "lazy" computation, combined with the minimal cost of determining the type of a pointer given its aliases, makes the additional cost of type analysis over pointer alias analysis insignificant.

For any point in a program, our analysis can answer the following question:

- What are the types of the objects that a pointer can point to?

We use this information to build the program call graph. In addition to resolving virtual function calls, this information can also be used at any pointer dereference, be it a data access, or a non-virtual function call. Where a pointer is passed as an argument to a function, the type of the object that the pointer points to can be used to *clone* the function based on the subtypes of the corresponding function parameter. This can increase opportunities for optimization within the cloned function body.

As in interprocedural pointer alias analysis, there exists a spectrum of solutions for the type inference problem, which can be characterized by their tradeoff in cost versus precision. An approach that computes the set of possible types from knowledge of the class hierarchy requires a single pass over the class definitions within the program, is the least costly, and has the least precision. An approach that computes the set of possible types by noting which classes have been instantiated requires a flow-insensitive traversal of the program, provides better precision than the previous approach, but does not utilize pointer alias information. An approach that computes the set of possible types by combining a flow-insensitive pointer alias analysis [2] with knowledge of the class hierarchy requires iteration over the program, and thus is more expensive than the previous two approaches, but also provides greater precision than these approaches. An approach that computes the set of possible types by combining a flow-sensitive pointer alias analysis [8, 3] with knowledge of the class hierarchy also requires iteration over the program, and thus is more expensive and more precise than the other approaches; this is the approach taken in this paper. Precise pointer alias information is required to perform several compiler optimizations on C++ programs. The flow-sensitive type analysis algorithm, presented in this paper, can be performed as part of the flow-sensitive alias analysis algorithm, at no incremental cost.

The simple example shown in Figure 1 illustrates the tradeoff between efficiency and precision of these approaches. At statement $m5$ of `main`, a method is invoked through the pointer `p`. By inspection of the source code, one can statically determine that method `D::f()` is invoked at this program point, since `p` will always point to an object whose class is `D`. The table in Figure 1 summarizes the set of methods that each approach determines may be invoked at statement $m5$, thereby illustrating the precision of the different approaches. The first column in the table lists the approaches. The remaining columns denote the possible `f()` methods that each approach determines can be invoked at $m5$.

The rest of the paper is organized as follows: Section 2 presents an overview of the pointer alias representation and the flow-sensitive pointer alias analysis algorithm in [8, 3]. Section 3 describes our adaptation of the flow-sensitive pointer alias algorithm to perform type analysis and a detailed example of its use. Section 4 presents a proof of the safety of the type analysis algorithm. Section 5 discusses how other C++ related issues are handled by the type analysis algorithm. Section 6 discusses prior work in the area of type analysis for object-oriented languages. Section 7 concludes this work.

2 Background

Our type analysis algorithm uses the pointer alias analysis framework of [8, 3] for imperative programming languages that include pointers. This section provides an overview of the pointer alias representation and the flow-sensitive interprocedural pointer alias algorithm described in [8, 3].

```
class A {
  virtual void f();
}
class B : public A {
  void f();
}
class C : public B {
  void f();
}
class D : public C {
  void f();
}
```

```
     main() {
m1:    B b;
m2:    A *p = new C;
m3:    p → f();
m4:    p = new D;
m5:    p → f();
     }
```

Approach	A::f()	B::f()	C::f()	D::f()
Class Hierarchy	X	X	X	X
Class Instantiation		X	X	X
Flow-Insensitive Pointer Alias			X	X
Flow-Sensitive Pointer Alias				X

Figure 1: Example C++ Program illustrating Precision

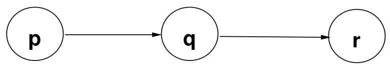

Figure 2: Example Alias Graph

2.1 Compact Representation

Access paths [20], which specify a way of accessing memory locations, are l-value expressions that may be constructed from variables, pointer indirection operators, and structure field selection operators. In C++ these l-value expressions include variables, the '$*$' dereference operator, the '.' field selection operator, and the '$\rightarrow$' operator, which is a combination of the '$*$' and the '.' operators. Aliasing occurs when two or more access paths refer to the same storage location. An *alias relation* (or *alias*), denoted $\langle *p, q\rangle$, signifies that access paths $*p$ and q are may-aliases at a program point. A *structural alias* arises from language constructs, such as the C union or Fortran equivalence statement. A structural alias remains constant for the entire program. A *reference parameter alias* can arise at a procedure call site, when a variable is passed by reference.

The method in [8, 3] assumes that every pointer memory location that is referenced has a unique name. Dynamically allocated objects are given a name corresponding to the statement in the program where they are created. An interprocedural naming technique is used – when a dynamically allocated object, which is involved in an alias relation, is propagated back to a calling routine, the call site information is appended to the object's name to distinguish different instances of the object that are allocated along different call paths [8, 3]. These memory locations are referred to as *named objects*.

A *compact representation* [8, 3] is created for pointer alias information by only allowing alias relations where at least one access path has no dereferencing, and the other access path has at most one level of dereferencing. The resulting alias relations determine a directed *alias graph*. Each named object that participates in an alias relation is associated with a single node in the alias graph. An outgoing edge represents a pointer dereference. Structures can be accommodated by augmenting the representation to distinguish different pointer fields of the same structure. Figure 2 illustrates an example alias graph. The edge from p to q represents the alias relation, $\langle *p, q\rangle$. The edge from q to r represents the alias relation, $\langle *q, r\rangle$. These two alias relations imply the alias relation $\langle **p, r\rangle$. Using the compact representation, the alias set for this graph is $\{\langle *p, q\rangle, \langle *q, r\rangle\}$, which implies the alias set $\{\langle *p, q\rangle, \langle *q, r\rangle, \langle **p, r\rangle, \langle **p, *q\rangle\}$.

Algorithm Alias-Analysis [3]

S_1: build the initial PCG
S_2: foreach procedure, P, in the PCG, loop
S_3: $Entry_P \leftarrow$ structural and reference parameter aliases for P
S_4: $Exit_P \leftarrow \{\}$
S_5: end loop
S_6: repeat
S_7:   foreach procedure P in the PCG, alternating between topological and reverse topological order, loop
S_8 repeat
S_9:   foreach SEG node, S (including $Exit_P$), loop
S_{10}:   compute flow sensitive intraprocedural aliasing:
S_{11}:   if S is a call site, C, calling procedure R then
S_{12}:   compute Out_C using $Exit_R$ and In_C
S_{13}:   else
S_{14}:   compute Out_S using In_S and its transfer function
S_{15}:   end loop
S_{16}:   until the intraprocedural alias sets for P converge
S_{17}:   foreach call site, C, calling procedure R loop
S_{18}:   update $Entry_R$, using In_C
S_{19}:   end loop
S_{20}:   end loop
S_{21}:   update the PCG using new function pointer aliases
S_{22}:   foreach new procedure P added to the PCG in step S_{21}, loop
S_{23}:   Initialize $Entry_P$ and $Exit_P$
S_{24}:   end loop
S_{25}: until the interprocedural alias sets converge

Figure 3: Flow-Sensitive Interprocedural Aliasing Algorithm

2.2 Flow-Sensitive Pointer Alias Algorithm

Through pointer assignments, a called procedure can affect the aliasing of a calling procedure at the point just after the call site. Similarly, the alias relations of the calling procedure can affect the aliases that are created in the called procedure. Thus, aliases are interprocedurally propagated from a call site to the entry point of a called procedure, and from the exit point of a called procedure to the point just after the call site. This propagation is made over the *program call graph* (PCG), which is a flow multigraph, where each node represents a unique procedure, and each edge (x, y) represents a possible invocation of procedure y from a call site located in procedure x.

Figure 3 repeats the algorithm presented in [3] for flow-sensitive interprocedural pointer alias analysis. This algorithm associates pointer alias information (represented as sets of aliases), with various points in the program. Each set is computed either during an intraprocedural phase (Steps S_8 – S_{16}) or an interprocedural phase (Steps S_{17} – S_{19}) of the framework, and thus is classified as either an intraprocedural or an interprocedural set.

The intraprocedural phase of the algorithm uses the Sparse Evaluation Graph (SEG) [9]. Nodes in this graph correspond to

1. nodes in the Control Flow Graph (CFG) of the procedure where pointer variables are potentially modified;
2. join nodes in the CFG where alias information is combined;
3. the *Entry* and *Exit* nodes of the CFG.

Details of the transfer function and the use of SEGs for pointer alias analysis can be found in [3].

A significant feature of the above algorithm is the *interleaving* of the intraprocedural and interprocedural phases. This interleaving removes the requirement that the iterations alternate topological directions when solving the bi-directional problem, although alternating directions may provide faster convergence. For this reason, we include this technique of alternating topological directions in our presentation of the algorithm.

3 Type Determination in C++

Consider a class `D`, which derives from a base class `B`. At run-time, when an object of class `D` is created, this object contains an object of class `B` as a subobject. Hence, even if a pointer variable p is declared to point to a `B` object, depending on further pointer assignments in the program, p could point to a complete `B` object or a `B` subobject in a `D` object. Thus, pointer alias information for p, i.e., the objects that p may point to, is necessary to infer the run-time type of $*p$.

Knowledge of the possible run-time types of objects is useful in resolving virtual function calls. Consider the example C++ program in Figure 4. At the virtual function call site $m6$, the function invoked at run-time is `D::createObj()`. Although `b` is *declared* to point to an object of type `B`, run-time type information shows that `B::createObj()` is not invoked at $m6$ since `b` points to a `D` object (actually, a `B` subobject in a `D` object). Similarly, at statement $m8$, `b` points to an `E` object, and therefore `E::createObj()` is invoked at that point.

The run-time types of objects can be used to improve the precision of alias information. For example, at statement $m6$, since we know the precise run-time type of $*b$, we know that the pointer variable `r` must point to the `D` object created and returned by `D::createObj`. Likewise, at statement $m9$, the sharper run-time type information for $*b$ allows us to assert that `*r` must be aliased to an `E` object. This illustrates how type information can improve the precision of pointer alias information, as well as any other interprocedural analysis.

Consider the virtual function call at statement $m11$. To ensure correctness in the absence of pointer alias information, we must assume conservatively that the type of `**p` is its declared type, `B`. However, to determine more precisely which method can be invoked at this statement, we need to know the aliases of `**p`. Pointer alias information [8, 3] can determine that `*p` is aliased to `*r`, and `*r` is aliased to both n_{d1} and n_{e1} (the dynamically allocated objects at statements $d1$ and $e1$) resulting from the merge point at statement $m10$. From this information we can determine that the types of objects aliased to `**p` are `D` and `E`, and not `B` or `C`. Since `E` inherits the implementation of `changeData` from `D`, the virtual function call at statement $m11$ can be converted to a direct call to function `D::changeData`. Thus, without pointer alias information, type information is either incorrect, or must conservatively assume that the type can be the declared type or any type inherited from the declared type.

Note that flow-sensitive interprocedural analysis is required to infer most of the alias and type information discussed above. Furthermore, a flow-sensitive interprocedural analysis can also be used to identify unreachable code. For example, in Figure 4, our analysis can determine that function `changeData(q)` defined in class `C` is never called, even though an object of type `C` is created in the program.

3.1 Approach

We augment alias information with two additional data structures: a type table and a class hierarchy graph. The *type table* is used to store the type information for all named dynamically allocated objects as well as statically allocated objects. The latter is available in the symbol table. Thus, all objects that can participate in an alias relation have an entry in the type table with their type.

We use the *Class Hierarchy Graph* (CHG) to help infer the types of objects that a pointer may point to. The CHG is defined as the tuple $\langle N, E \rangle$, where N is the set of classes in the C++ program, and E is the set of edges representing inheritance relationship between classes. Since not all classes of a C++ program may be related by the inheritance relation, a CHG for a program can potentially be a forest of DAGs.

Pointer alias information gives the set of named objects, A_p, that a pointer p may point to. While computing the run-time types of $*p$, we only consider those named objects in A_p whose types belong to the subgraph of the CHG rooted at the declared type of $*p$ [12]. The types of these named objects are retrieved from the type table. The algorithm to retrieve the possible run-time types of $*p$, given the aliases of $*p$, is

```
    class B {
      int data;
      virtual B *createObj();
      virtual void changeData(B &q);
    };

    B * B::createObj() {
b1:   return(new B);
    }
    B::changeData(B &q) {
b2:   q.data = 40;
    }
```

```
    class C : public B {
      int Cdata;
      void changeData(B &q);
    }

    C::changeData(B &q) {
c1:   q.data = 50;
    }
```

```
    class D : public B {
      B *createObj();
      void changeData(B &q);
    };

    B* D::createObj() {
d1:   return(new D);
    }
    D::changeData(B &q) {
d2:   q.data = 20;
    }
```

```
    class E : public D {
      B *createObj();
    };

    B* E::createObj() {
e1:   return(new E);
    }
```

```
      main() {
m1:   B q, *r, *b, **p;
m2:   C c;
m3:   p = &r;
m4:   if (cond1) {
m5:     b = new D;
m6:     r = b → createObj();
m7:   } else {
m8:     b = new E;
m9:     r = b → createObj();
m10:  }
m11:  (*p) → changeData(q);
m12:  cout << q.data;
      }
```

Figure 4: Example C++ Program

Algorithm Retrieve-Types(p, A_p)

```
/* p is a pointer variable.
   A_p is the set of named objects aliased to *p. */

let T_d be the declared type of p (from type table)
let G be the CHG that is rooted at T_d
let Result be a set of types
Result ← {}
foreach n ∈ A_p, loop
    let t be the type of n from the type table
    if t ∈ G then
        Result ← Result ∪ {t}
    end if
end loop
return Result /* the possible run-time types of *p */
```

Figure 5: Algorithm to Retrieve the Run-Time Types, Given Alias Information

presented in Algorithm **Retrieve-Types** in Figure 5. The algorithm takes as input the pointer variable, p, and the aliases of $*p$, A_p.

In C++, dynamic dispatch of virtual functions can also be obtained by using reference parameters. To retrieve the possible run-time types of C++ reference parameters, a similar analysis to the one we present in this paper is necessary. In particular, the **Retrieve-Types** algorithm would also require reference parameter alias information.

By adapting the pointer alias analysis algorithm of [8, 3] for each pointer, p, we will have the set of named objects, and their types, that p may point to (is aliased with) at each program point. As in [8, 3], our algorithm is optimistic in that we assume a pointer does not point to anything initially. As statements are processed, pointer aliases can be generated and killed via pointer assignment statements.

3.2 Algorithm

The type analysis algorithm in Figure 6 proceeds as in [3], interleaving the interprocedural and intraprocedural phases. The algorithm terminates when the pointer alias information reaches a fixed point. The type information for a pointer at any point in the program can be computed using Algorithm **Retrieve-Types**. Note that our algorithm is essentially the same as the corresponding pointer alias analysis algorithm in Figure 3, except for the type table update and use of the type information in resolving virtual function calls. At a virtual function call site, $p \rightarrow foo()$, the PCG is not complete until the type of each of the objects potentially pointed to by p is known. Since the static type of all objects is known, this amounts to knowing the objects that a pointer points to. The next section discusses the analysis of virtual function calls.

3.3 Handling Virtual Function Calls

Our scheme for analyzing virtual functions is based on the handling of function pointers presented in [2, 3]. We build the PCG incrementally as the algorithm infers the possible types of pointer variables. If p is a pointer that is used to invoke a virtual method $p \rightarrow foo()$, we use the approach in Section 3.1 to infer the types of objects that p may point to. The run-time types of $*p$ determine which methods can be invoked through p. We build the PCG for virtual function invocations using the algorithm in Figure 7. Note that this algorithm does not add a PCG edge to a pure virtual function definition. Such function definitions have no implementations and can never be invoked at run-time.

Our algorithm for constructing the PCG extends the algorithm presented in [2, 3] for function pointers by using properties specific to C++. Although not required for correctness, the use of these properties improves

Algorithm Type-Analysis

S_1: build the initial PCG, ignoring virtual function and function pointer call sites
S_2: foreach function, P, in the PCG, loop
S_3: $Entry_P \leftarrow$ structural and reference parameter aliases for P
S_4: $Exit_P \leftarrow \{\}$
S_5: end loop
S_6: repeat
S_7: foreach function P in the PCG, alternating between topological and reverse topological order, loop
S_8: if this is the first time P has been processed,
initialize the type table with all objects potentially created in P.
S_9 repeat
S_{10}: foreach SEG node, S (including $Exit_P$), loop
S_{11}: compute flow sensitive intraprocedural aliasing:
S_{12}: if S is a call site, C, calling function R then
S_{13}: compute Out_C using $Exit_R$ and In_C
S_{14}: else
S_{15}: compute Out_S using In_S and S's transfer function
S_{16}: end if
S_{17}: end loop
S_{18}: until the intraprocedural alias sets for P converge
S_{19}: foreach call site, C in P, loop
S_{20}: if C is a virtual function call site then
S_{21}: update the PCG using pointer aliases in In_C
and Algorithm **PCG-Virtual-Function-Update** in Figure 7.
S_{22}: end if
S_{23}: if C is a function pointer call site then
S_{24}: update the PCG using function pointer aliases in In_c
S_{25}: end if
S_{26}: foreach newly added function to the PCG, R, loop
S_{27}: Initialize $Entry_R$ and $Exit_R$ as in S_3 and S_4
S_{28}: end loop
S_{29}: foreach PCG edge, $P \rightarrow R$, at call site C, loop
S_{30}: update $Entry_R$, using In_C
S_{31}: end loop
S_{33}: end loop
S_{34}: end loop
S_{35}: until interprocedural alias sets converge

Figure 6: Interprocedural Type Analysis Algorithm

Algorithm PCG-Virtual-Function-Update

Let C be the call site that invokes $p \rightarrow$ `foo()` in function P.
Let A_p be the aliases of $*p$ at C
/* Compute the set of inferred types of $*p$ at C */
$\mathcal{T} \leftarrow$ **Retrieve-Types**(p, A_p)
Add a PCG edge from calling function P, with label C, to all non-pure virtual functions `foo()` that are defined in, or inherited by, classes in the set, $\mathcal{T}$.

Figure 7: Algorithm to Update PCG at Virtual Function Call Sites

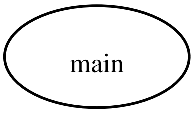

Figure 8: Initial *PCG*

the precision of the pointer alias and type information.

3.4 Example

This subsection provides a detailed description of our type analysis algorithm for the example in Figure 4. Figure 8 presents the initial PCG. During the initial construction of the PCG, virtual function call sites are ignored. Figures 9–12 present the results (alias relations, PCG, and type table) of each iteration of the outer loop of our algorithm. For simplicity, we do not use the interprocedural naming technique, mentioned in Section 2, while illustrating this example. The alias relations for `main` are those that hold after the statement has executed. The outer loop of our type analysis algorithm alternates topological directions, which results in the algorithm converging after four iterations for this example program.

Iteration 1

Figure 9 illustrates the results after the first iteration. The first table is the type table which contains information from `main`. The second table contains the alias sets at various points in the program. Since the *Exit* set of functions not yet present in the PCG are assumed to be empty, the alias sets at the return point of each call site in `main` $(m6, m9, m11)$ are also empty. After the alias information for `main` is computed, the alias relations at the beginning of statement $m6$ (the alias relations after $m5$) and the **Retrieve-Types** algorithm are used to determine that `b` may point to an object (n_{m5}) of type `D` at $m6$. Hence, `D::createObj()` may be called by that virtual function invocation. Likewise, the alias relations at the beginning of statement $m9$ are used to determine that a method of an object (n_{m8}) of type `E` may be called by that virtual function invocation. Thus, the PCG is updated (step S_{24}) to reflect that the virtual function calls at statements $m6$ and $m9$ may call `D::createObj`, and `E::createObj`, respectively. At the beginning of statement $m11$, aliases of $**p$ have yet to be determined. Thus, the types of the objects that invoke the virtual functions cannot yet be inferred. Figure 9 shows the updated PCG with edges labeled with call site information. The *Entry* sets (in the alias table) for functions `D::createObj` and `E::createObj` are initialized with alias information from $m6$ and $m9$, respectively, by steps S_{27} and S_{30} of our algorithm. The *Exit* sets for these functions are also initialized by step S_{29}.

Iteration 2

The updated PCG from the first iteration is used during the second iteration, which proceeds in reverse topological order. During this iteration, three functions are visited: `E::createObj`, `D::createObj`, and `main`. Figure 10 illustrates the results after the second iteration.

Function `E::createObj` is visited, and the value of $Entry_{E::CreateObj}$, which contains aliases propagated to the call site at $m9$, is used to compute $Exit_{E::CreateObj}$. This set contains these aliases and the new alias generated in this function, $\langle *ret, n_{e1}\rangle$, where ret is a representation of the return value of the function [3], which will be mapped to the appropriate variable at all call sites.

Next, function `D::createObj` is visited, and value of $Exit_{D::createObj}$ is computed in a similar manner.

Finally, `main` is visited, and the exit sets just computed are incorporated at $m6$ and $m9$, and propagated forward to $m10$.

After alias information converges, the alias relations $\langle *p, r\rangle$ and $\langle *r, n_{d1}\rangle$, and $\langle *p, r\rangle$ and $\langle *r, n_{e1}\rangle$ holding at the beginning of statement $m11$ are used to determine that a method of an object of type `D` or `E` $(n_{d1}$ or $n_{e1})$ may be called by that virtual function invocation. Since both classes have the same implementation of this method, `D::changeData`, the PCG is updated to reflect the virtual function call at statement $m11$ from `main` to `D::changeData`. After `D::changeData` is added to the PCG, $Entry_{D::changeData}$ is initialized with the alias information from $m11$.

Object	Type
q	B
r	B*
b	B*
p	B**
c	C
n_{m5}	D
n_{m8}	E

Stmt	Alias Relations
$m3$	$\langle *p, r \rangle$
$m4$	$\langle *p, r \rangle$
$m5$	$\langle *p, r \rangle, \langle *b, n_{m5} \rangle$
$m6$	$\{\}$
$m7$	$\langle *p, r \rangle$
$m8$	$\langle *p, r \rangle, \langle *b, n_{m8} \rangle$
$m9$	$\{\}$
$m10$	$\{\}$
$m11$	$\{\}$
$m12$	$\{\}$
$Entry_{D::createObj}$	$\langle *p, r \rangle, \langle *b, n_{m5} \rangle$
$Exit_{D::createObj}$	$\{\}$
$Entry_{E::createObj}$	$\langle *p, r \rangle, \langle *b, n_{m8} \rangle$
$Exit_{E::createObj}$	$\{\}$

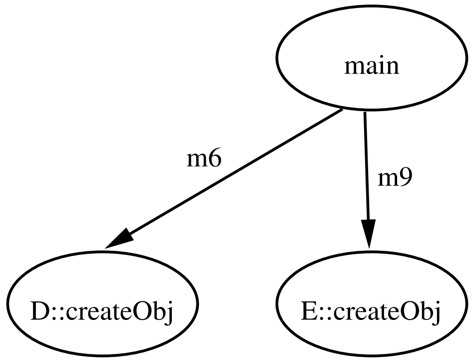

Figure 9: The Type Table, Alias Relations, and *PCG* after the 1st Iteration

Object	Type
q	B
r	B*
b	B*
p	B**
c	C
n_{m5}	D
n_{m8}	E
n_{d1}	D
n_{e1}	E

Stmt	Alias Relations
$m3$	$\langle *p, r\rangle$
$m4$	$\langle *p, r\rangle$
$m5$	$\langle *p, r\rangle$, $\langle *b, n_{m5}\rangle$
$m6$	$\langle *p, r\rangle$, $\langle *b, n_{m5}\rangle$, $\langle *r, n_{d1}\rangle$
$m7$	$\langle *p, r\rangle$
$m8$	$\langle *p, r\rangle$, $\langle *b, n_{m8}\rangle$
$m9$	$\langle *p, r\rangle$, $\langle *b, n_{m8}\rangle$, $\langle *r, n_{e1}\rangle$
$m10$	$\langle *p, r\rangle$, $\langle *b, n_{m5}\rangle$, $\langle *r, n_{d1}\rangle$, $\langle *b, n_{m8}\rangle$, $\langle *r, n_{e1}\rangle$
$m11$	$\{\}$
$m12$	$\{\}$
$Entry_{D::createObj}$	$\langle *p, r\rangle$, $\langle *b, n_{m5}\rangle$
$Exit_{D::createObj}$	$\langle *p, r\rangle$, $\langle *b, n_{m5}\rangle$, $\langle *ret, n_{d1}\rangle$
$Entry_{E::createObj}$	$\langle *p, r\rangle$, $\langle *b, n_{m8}\rangle$
$Exit_{E::createObj}$	$\langle *p, r\rangle$, $\langle *b, n_{m8}\rangle$, $\langle *ret, n_{e1}\rangle$
$Entry_{D::changeData}$	$\langle *p, r\rangle$, $\langle *b, n_{m5}\rangle$, $\langle *r, n_{d1}\rangle$, $\langle *b, n_{m8}\rangle$, $\langle *r, n_{e1}\rangle$
$Exit_{D::changeData}$	$\{\}$

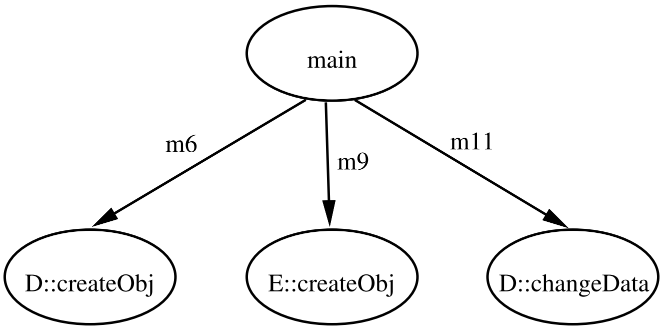

Figure 10: The (final) Type Table, Alias Relations, and (final) *PCG* after the 2nd iteration

Stmt	Alias Relations
$m3$	$\langle *p, r\rangle$
$m4$	$\langle *p, r\rangle$
$m5$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle$
$m6$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle$
$m7$	$\langle *p, r\rangle$
$m8$	$\langle *p, r\rangle, \langle *b, n_{m8}\rangle$
$m9$	$\langle *p, r\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$
$m10$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$
$m11$	$\{\}$
$m12$	$\{\}$
$Entry_{D::createObj}$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle$
$Exit_{D::createObj}$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *ret, n_{d1}\rangle$
$Entry_{E::createObj}$	$\langle *p, r\rangle, \langle *b, n_{m8}\rangle$
$Exit_{E::createObj}$	$\langle *p, r\rangle, \langle *b, n_{m8}\rangle, \langle *ret, n_{e1}\rangle$
$Entry_{D::changeData}$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$
$Exit_{D::changeData}$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$

Figure 11: Alias Relations after the 3rd Iteration

Iteration 3

Four functions are visited in topological order during the third iteration: `main`, `D::createObj`, `E::createObj`, and `D::changeData`. The alias relations in the first three functions do not change. However, since this is the initial visit to `D::changeData`, the alias information in $Entry_{D::changeData}$ is propagated to $Exit_{D::changeData}$. No update to the PCG is made. Figure 11 illustrates the results after the third iteration.

Iteration 4

The fourth iteration proceeds in reverse topological order as is given in Figure 12. The only change is to statements $m11$ and $m12$ in `main`, where the alias information from $Exit_{D::changeData}$ is propagated back to $m11$ and on to $m12$.

The final PCG (Figure 10) only contains nodes and edges for functions and potential invocations. Functions `B::createObj`, `B::changeData`, and `C::changeData` are never invoked, and thus not part of the PCG. Consequently, these functions do not need to be analyzed or included in the final program. From the final PCG, it can be inferred that all virtual function call sites invoke exactly one function. Hence, these call sites can be converted to direct function calls.

4 Proof of the Algorithm

This section presents a proof of the safety of the type analysis algorithm. We first prove the following lemma:

Lemma 1 *If pointer alias information for $*p$ is safe at a point, s, in the program, then the type information computed by Algorithm* **Retrieve-Types** *for $*p$ is safe.*

Proof Assume the pointer alias information for $*p$ is safe at program point s. Thus, all objects that p may point to are known at this point. Since the pointer alias representation we employ names all objects, all aliases of $*p$ are named. Further, as C++ is a statically typed language, the type of each object is available at compile time and is stored in the type table. By the semantics of C++ [12], p must point to a type rooted at the declared type of p in the CHG. Thus, only these types are required.

Hence, the type information for $*p$ computed by Algorithm **Retrieve-Types** is safe. ¶

Stmt	Alias Relations
$m3$	$\langle *p, r\rangle$
$m4$	$\langle *p, r\rangle$
$m5$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle$
$m6$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle$
$m7$	$\langle *p, r\rangle$
$m8$	$\langle *p, r\rangle, \langle *b, n_{m8}\rangle$
$m9$	$\langle *p, r\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$
$m10$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$
$m11$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$
$m12$	$\langle *p, r\rangle, \langle *b, n_{m3}\rangle, \langle *r, n_{d1}\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$
$Entry_{D::createObj}$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle$
$Exit_{D::createObj}$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *ret, n_{d1}\rangle$
$Entry_{E::createObj}$	$\langle *p, r\rangle, \langle *b, n_{m8}\rangle$
$Exit_{E::createObj}$	$\langle *p, r\rangle, \langle *b, n_{m8}\rangle, \langle *ret, n_{e1}\rangle$
$Entry_{D::changeData}$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$
$Exit_{D::changeData}$	$\langle *p, r\rangle, \langle *b, n_{m5}\rangle, \langle *r, n_{d1}\rangle, \langle *b, n_{m8}\rangle, \langle *r, n_{e1}\rangle$

Figure 12: Alias Relations after the Final Iteration

We use the notion of an execution trace in the proof of the safety of the type analysis algorithm. An *execution trace* is a sequence of program points that represents the run-time execution order of the statements of a program for a particular execution. For the purposes of the proof, we assume that each program point in the execution trace of a function corresponds to a single statement of the program. We represent the execution trace by a simple graph called the *Execution Trace Graph*, where each node represents a program point, and has *at most* one predecessor.

Our type analysis algorithm focuses on computing a conservative approximation to the run-time types of $*p$, for all pointer variables p in a C++ program. In our proof, we show that the types for $*p$ computed by our algorithm at any point in the program includes the actual run-time type of $*p$ at that point.

Definition 1 *An Execution Trace Graph is a triple* $\langle N_e, E_e, Start\rangle$, *where*

- N_e *is the set of nodes; each node in* N_e *is labeled as a pair* $\langle F, n\rangle$*; where* F *represents a function node in the PCG; and* n *represents a statement in the execution trace for* F*. In addition, for every call site,* $\langle F, c\rangle \in N_e$*, in function* F*, we associate a return point node* $\langle F, r\rangle \in N_e$*.* F *and* n *together identify a specific program point within the program.*
- E_e *is the set of edges; each edge* $e = (p, q) \in E_e$ *represents either interprocedural or intraprocedural control flow from node* p *to node* q*.*
- *Start is the unique start node of the program.*

Definition 2 *We define an execution path* $\mathcal{P}(\langle F, n\rangle)$ *of length* l *for a node labeled* $\langle F, n\rangle$ *in the Execution Trace Graph as the path of length* l *from Start up to and including this node.*

We prove a lemma that shows that our algorithm for updating the *PCG* at virtual function call sites is a conservative approximation of the actual functions invoked at such call sites, for any execution of the C++ program.

Lemma 2 *For a given execution of the program, at a virtual function call site,* $c : p \rightarrow \mathcal{F}$*, if the run-time type of* $*p$ *is in the set of inferred types of* $*p$*, a PCG edge from the calling function to the actual function invoked at* c *will be added by Algorithm* **PCG-Virtual-Function-Update**.

Proof Assume that the run-time type of $*p$, X, belongs to the set of inferred type(s) of $*p$ at the virtual function call site c, for some execution of the program. Assume $\mathcal{F}_r$ is the function invoked at run-time via the pointer p. Thus, $\mathcal{F}_r$ must be defined in or inherited by X [12]. Let $\mathcal{G}$ be the calling function that contains call site c. Algorithm **PCG-Virtual-Function-Update** adds a PCG edge labeled c from $\mathcal{G}$ to all the functions $\mathcal{F}$ that are defined in, or inherited by, classes in the inferred types of $*p$. Since X belongs to the inferred types of $*p$, a PCG edge labeled c must be added to $\mathcal{F}_r$ by this algorithm.

¶

The following theorem states the safety of the type analysis algorithm. Its proof uses the results from Lemmas 1 and 2 and the safety of the algorithm in [3] to compute pointer alias information.

Theorem 1 *Algorithm* **Type-Analysis** *in Figure 6 computes safe run-time type information at any point in a* C++ *program.*

Proof We first claim that the algorithm computes safe type information for any given execution $\mathcal{E}$ of the program. We prove this claim by induction on the length of execution path, $\mathcal{P}$. We show in each case that the run-time types at a program point for this execution of the program is a subset of the types inferred at the same program point by Algorithm **Type-Analysis**.

Base Case:

The base case corresponds to the *Start* node in the Execution Trace Graph for this execution of the program. Hence, there are no executable statements before this node. In addition, the *Start* node does not represent an executable statement. Therefore, no objects are created along this execution path. Hence, there are no run-time types of $*p$, for all pointer variables, p.

Induction Hypothesis:

Assume that for execution $\mathcal{E}$ of the program, our type analysis algorithm computes safe type information for all nodes $\langle F, n\rangle$, such that $\mathcal{P}(\langle F, n\rangle)$ is a path of length $L-1$.

Induction Step:

Consider the node, $\langle F', n'\rangle$, whose path length, $\mathcal{P}(\langle F', n'\rangle)$, is L. From the induction hypothesis, we have safe type information at the predecessor of the node labeled $\langle F', n'\rangle$. We refer to this node as $\langle F_p, n_p\rangle$. The proof of the induction step depends on the type of node, $\langle F', n'\rangle$. In the following discussion, we distinguish between an entry point of a virtual function and all other program points.

When $\langle F', n'\rangle$ corresponds to the entry point of a virtual function, the predecessor node, $\langle F_p, n_p\rangle$, is a virtual function call site. From induction hypothesis, we know that for execution $\mathcal{E}$ of the program, the type information inferred at $\langle F_p, n_p\rangle$ is safe. Hence from Lemma 2, the PCG update at $\langle F_p, n_p\rangle$ is safe for this execution of the program. The safety of the type information at $\langle F', n'\rangle$ follows from the safety of the data flow equations and algorithms in [8, 3] and Lemma 1.

The proof of the induction step when $\langle F', n'\rangle$ is any other program point other than a virtual function entry point follows directly from the induction hypothesis (i.e., the safety of the type information at $\langle F_p, n_p\rangle$), Lemma 1, and the safety of the data flow equations and algorithms in [8, 3].

Thus, the set of run-time types of objects at any program point for some execution of a C++ program is a subset of the set of types of objects inferred for the program by Algorithm **Type-Analysis**. Since we have shown safety for an arbitrary program execution, the safety for *all* program executions follows. ¶

5 Other C++ Related Issues

C++ uses the implicit `this` pointer and also invokes implicit constructor and destructor functions. In the next two subsections, we indicate how our analysis algorithm accommodates these features.

5.1 Handling the this Pointer

The `this` pointer is an object pointer that is implicitly passed at function call sites and used within method bodies. The `this` pointer points to the object that defines the invoked (virtual or non-virtual) methods. Since a `this` pointer can also be used implicitly, it is desirable for the analysis to pass the type of `*this` to the called method at function invocation points. If the function invoked is a non-virtual method, the type of `*this` can be trivially inferred from the class in which the method is defined. Inferring the type of `*this` at a virtual function call site requires more analysis. The type of `*this` can be used to resolve virtual function calls within the called method.

In the case of a virtual function call, $p \rightarrow foo()$, our algorithm updates the PCG using the pointer alias information for $*p$, A_p, the type table information, and the CHG. Suppose a PCG edge is created from the function invoking this virtual function call to function `X::foo()`. The aliases for `*this` used in `X::foo()` is computed using A_p and the CHG rooted at `X` as follows: we explicitly alias `*this` to those objects in A_p that p points to such that the types of these objects correspond to classes in the CHG rooted at `X`. The type(s) of `*this` can be inferred using the type table.

In the case of a direct function call, $q.foo()$, where q has a static type, we create the alias, $\langle *this, q \rangle$.

Since an assignment to a `this` pointer is never allowed, it suffices to perform flow-insensitive intraprocedural alias analysis on such pointers. However, we propagate information interprocedurally during the flow sensitive analysis.

5.2 Handling Constructors and Destructors

Constructors and destructors can be implicitly called. We note that when a "new" is invoked, the appropriate constructor for the class is invoked. In addition, the constructor for all the base classes and embedded members of class types must be invoked before the constructor of the current object. Note that the `this` pointer (pointing to the current object) must be passed to the base and embedded object constructor functions. We make these implicit function calls explicit for our analysis, add them to the PCG, and create `this` aliases as described in the previous section.

Invocation of destructors occurs either explicitly or during a "delete" invocation. Issues similar to those of constructors are also present here, and we handle them in an analogous way. Further, since destructors can be virtual, the type(s) of the object the pointer points to is used to identify the corresponding virtual destructor(s) (as in any other virtual function call).

6 Related Work

Pande and Ryder [22, 24] presented the first algorithm for determining types in the presence of pointer aliases for C++. They extend a previously developed pointer alias analysis algorithm [18, 19] to compute type information. Pande and Ryder formulate their type analysis and alias analysis as part of a single data flow framework with a function defined at each *ICFG* node that includes both *points_to* and *may_hold* predicates. Their method computes an exhaustive representation for both alias and type information at each program point. Although we also extend a previously developed pointer alias analysis algorithm [8, 3] to compute type information, our approach differs from Pande and Ryder in two fundamental ways. First, we use a compact representation which defers the computation of aliases for pointers with multiple levels of indirection until such aliases are needed. Second, the computation of type information is performed in a "lazy" manner, i.e., until such information is needed. Pande and Ryder present preliminary results for a limited program model (single inheritance and non-recursive data structures) [22]. Support for non-recursive data structures and additional results are presented in [23]. These results suggest that a flow-sensitive interprocedural type analysis technique can eliminate a significant number of functions as possible receivers at virtual function call sites.

Pande and Ryder describe a technique to refine alias and type information propagated to virtual function calls by removing information that could not hold given that the call was made. This technique can be used directly by our algorithm when the call is made via a single-level dereference ("p→foo()"). However, in the compact representation multiple-level pointers require the combination of alias relations. Consequently, to apply the Pande-Ryder technique for virtual function calls using multiple-level pointers, the corresponding single-level alias relations could be recorded and used to disallow a subsequent combination of these components in the called routine. Techniques to detect the invalid combinations of aliases using the compact representation are given in [3].

The technique of Pande and Ryder mention above can be extended to handle function pointer aliases. For example, a call through a function pointer, "*p()", should not propagate the alias $\langle *p, f\rangle$ to function g because its holding contradicts f being invoked.

Palsberg and Schwartzbach [21] propose a constraint-based type analysis algorithm that supports object-oriented languages with late binding, inheritance, and assignments. This algorithm has a worst case running time of $O(N^3)$. However, the algorithm does not consider control flow and aliases introduced by pointer assignments in the program. Kumar et. al. [17] have extended the Palsberg/Scwharzbach approach to consider control flow, by using a Static Single Assignment Form based approach. However, their algorithm does not accommodate pointer aliases either. Both these approaches are applicable to dynamically typed object-oriented languages.

Plevyak and Chien [26] describe a type inference algorithm for the dynamically-typed concurrent object-oriented language, Concurrent Aggregates. Their algorithm uses abstract interpretation and the constraint based technique of Palsberg and Schwartzbach. Although this algorithm handles interprocedural control flow, pointer aliases are not addressed.

Bacon [1] describes a flow-insensitive algorithm that keeps track of the global set of classes that may be instantiated. A virtual function call is added to the PCG if the class associated with the method has been instantiated anywhere else in the reachable code of the PCG. This algorithm is linear in the size of the PCG, but since it does not consider pointer alias information or control flow, it can be less precise than our method.

Diwan, Moss, and McKinley [11] describe methods for converting method invocations to direct calls in Modula-3. These methods include type hierarchy analysis, interprocedural and intraprocedural type propagation for scalars, *self-null* analysis which takes advantage of the fact that the receiver cannot be NULL within a method, and a simple form of heap analysis which does not distinguish different instances of the same object. Since only scalar types are propagated and Modula-3 semantics restrict pointer aliases, no pointer alias analysis is used with these methods. Results are presented which suggest that more sophisticated interprocedural analysis is unlikely to be beneficial for Modula-3.

Fernandez [13] uses knowledge of the class hierarchy to replace virtual method invocations with direct calls in Modula-3 programs. This optimization is performed at link time, on a machine-independent intermediate code.

Dean, Chambers, and Grove [10] describe a technique for function *specialization* in Cecil that combines knowledge of the class hierarchy with run-time profiling information.

Steensgaard [28] describes a method for performing an interprocedural, flow-insensitive, points-to alias analysis based on type inference methods with an almost linear time cost complexity. He describes his method as obtaining results equivalent to those obtained by performing a flow-insensitive alias analysis which assumes that alias relations are reflexive and transitive.

Kuhn and Binkley [16] describe an algorithm for improving the execution performance of C++ by replacing a virtual function call with a switch statement that contains a case for each possible method invocation. As no type analysis is performed, the full class hierarchy is used to determine the set of possible methods that can be invoked. Language semantics are used to eliminate pure virtual functions.

Chambers and Ungar [7] propose an iterative type analysis algorithm for the dynamically-typed object-oriented language, SELF. Several optimizations have been implemented in the SELF compiler project to reduce the overhead of dynamic dispatches in SELF programs. These include message splitting, uncommon branch elimination, customization, run-time type feedback, and inline caching [14, 15, 7, 6]. Most of these optimization techniques can be applied to programs written in C++ as well. The types of objects computed at compile-time by the algorithm presented in this paper will greatly enhance the benefits of these optimizations.

Since C++ is a statically typed language, a number of issues that arise while inferring types in dynamically-typed languages do not arise in the context of C++. However, as noted above, most of the prior work on inferring types for dynamically-typed languages are not concerned with pointer aliases.

7 Conclusions

Run-time types in C++ arise through the use of pointers. Thus, knowledge of pointer aliases improves the precision of type analysis. In this paper we have presented an algorithm for computing program point specific type information for static object-oriented languages, such as C++, which uses alias information. We have shown that the compact representation permits type analysis to be a natural extension of an existing flow-sensitive pointer alias analysis algorithm. Since our type analysis algorithm is an extension of this existing algorithm, type analysis can be performed with little or no increased cost over the underlying pointer alias analysis. We have shown that our method can also improve the precision of the pointer alias analysis, and demonstrated that the precision of our method can be improved by the incorporation of language semantics and the class hierarchy graph. The availability of type information and the resolution of virtual function calls permits a wide range of interprocedural analyses and transformations, including inlining and cloning.

Acknowledgements

We thank David Bacon, Yong-fong Lee, Lee Nackman, Hemant Pande, Ganesan Ramalingam, Peter Sweeney, and Laureen Treacy for their suggestions for improving earlier drafts of this paper.

References

[1] David F. Bacon. Rapid type inference for C++ programs. Technical report, IBM T.J. Watson Research Center, February 1995.

[2] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, 892*, pages 234–250. Springer-Verlag, 1995. Proceedings from the *7th International Workshop on Languages and Compilers for Parallel Computing.* Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September, 1994.

[3] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Interprocedural pointer alias analysis. Technical report, IBM T.J. Watson Research Center, August 1995.

[4] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *21st Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 397–408, Portland, OR, January 1994.

[5] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 1995.

[6] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

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

[8] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induces aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, January 1993.

[9] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *18th Annual ACM Symposium on the Principles of Programming Languages*, January 1991.

[10] Jeffery Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 93–101, June 1995.

[11] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Analyzing statically-typed object-oriented programs for modern processors. Technical Report available via http://www.cs.umass.edu, University of Massachusetts, July 1995.

[12] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[13] Mary F. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, June 1995.

[14] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*, July 1991.

[15] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[16] Bradley M. Kuhn and David W. Binkley. An enabling optimization for C++ virtual functions. In *Mid-Atlantic Graduate Workshop on Programming Languages and Systems*, April 1995.

[17] Sandeep Kumar, Dharma P. Agrawal, and S. Purushothaman Iyer. An Improved Type-Inference Algorithm To Expose Parallelism In Object-Oriented Programs. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 283–286. Kluwer Academic, Boston, MA, 1995.

[18] William Landi and Barbara Ryder. Pointer-induced aliasing: A problem classification. In *18th Annual ACM Symposium on the Principles of Programming Languages*, pages 93–108, January 1991.

[19] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[20] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[21] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems.* John-Wiley & Sons Ltd., 1994.

[22] Hemant Pande and Barbara Ryder. Static type determination and aliasing for C++. Technical Report LCSR-TR-236, Rutgers University, January 1995.

[23] Hemant Pande and Barbara Ryder. Static type determination and aliasing for C++. Technical Report LCSR-TR-250, Rutgers University, July 1995.

[24] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. In *Sixth Usenix C++ Conference*, 1994.

[25] Ramesh Parameswaran. Interprocedural alias and type analysis for pointers. M.S. Thesis, Department of Computer Science, University of Wisconsin–Madison, 1992.

[26] John Plevyak and Andrew A. Chien. Precise object-oriented type inference and its use in program optimization. Technical report, Univ. of Illinois, Urbana-Champaign, October 1994.

[27] G. Ramalingam. The undecidability of type analysis. Unpublished Manuscript, July 1995.

[28] Bjarne Steensgaard. Points-to analysis in almost linear time. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, 1996. preliminary draft.