Simple and Effective Analysis of Statically-Typed Object-Oriented Programs

Amer Diwan J. Eliot B. Moss Kathryn S. McKinley*

Department of Computer Science
University of Massachusetts, Amherst, MA 01003

Abstract

To use modern hardware effectively, compilers need extensive control-flow information. Unfortunately, the frequent method invocations in object-oriented languages obscure control flow. In this paper, we describe and evaluate a range of analysis techniques to convert method invocations into direct calls for statically-typed object-oriented languages and thus improve control-flow information in object-oriented languages. We present simple algorithms for *type hierarchy analysis*, *aggregate analysis*, and *interprocedural and intraprocedural type propagation.* These algorithms are also fast, $O(|\text{procedures}| * \sum_{p}^{procedures} n_p * v_p)$ worst case time (linear in practice) for our slowest analysis, where n_p is the size of procedure p and v_p is the number of variables in procedure p, and are thus practical for use in a compiler. When they fail, we introduce *cause analysis* to reveal the source of imprecision and suggest where more powerful algorithms may be warranted. We show that our simple analyses perform almost as well as an oracle that resolves all method invocations that invoke only a single procedure.

*The authors can be reached electronically via Internet addresses {diwan,moss,mckinley}@cs.umass.edu. This work was supported by the National Science Foundation under grants CCR-9211272 and CCR-9525767 and by gifts from Sun Microsystems Laboratories, Inc.

To appear in OOPSLA 96: Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications.

1 Introduction

Computer hardware and programming languages are on a collision course. Modern hardware exposes an increasing amount of implementation details to the compiler. The exposed detail includes the pipeline, the memory hierarchy, and the functional parallelism of the processor. To achieve the processor's potential, the compiler must be able to predict the program's control-flow.

In object-oriented languages, programmers use a type hierarchy and method invocations to improve code reuse and correctness. Unfortunately, method invocations obscure which procedure is actually being invoked. In dynamically-typed languages, frequent method look-up is costly in itself [6] but in statically-typed languages, it is typically not a significant cost. For both static and dynamic languages however, method invocations inhibit optimization. If analysis can resolve method invocations to direct calls, the compiler can then optimize effectively across method invocation sites, replacing the method invocation with a direct call, a tailored call, or an inlined call. The additional control-flow information provides fodder for an optimizing compiler to improve performance.

In this paper, we describe and evaluate the following analyses for resolving method invocations in a statically-typed object-oriented language:

- *Type hierarchy analysis* determines the procedures a method invocation may call by considering the types that implement that method.
- *Type propagation* performs intraprocedural and interprocedural data-flow analysis for types.
- *Aggregate analysis* detects when an object or

record field is restricted to a single type.

Our type hierarchy analysis is the same as previous work. What differentiates our type propagation and aggregate analysis from previous work is that we make simplifications to achieve a fast whole program analysis, $O(|\text{procedures}| * \sum_p^{procedures} n_p * v_p)$ time worst case (linear in practice) for the slowest analysis, where n_p is the size of procedure p and v_p is the number of variables in procedure p. We also introduce a *cause analysis* algorithm that determines the reason when our analysis does not resolve a method invocation. Using the cause analysis, we present detailed experimental results that characterize which language and programming features affect the procedure(s) called at run-time from a method invocation site and the ability of the analysis to resolve the method invocations.

We implemented the analyses for Modula-3 programs [19] in the SRC Modula-3 compiler version 3.5 [16]. Modula-3 is a statically-typed object-oriented systems programming language similar (for present purposes) to C++ and even more so to Java. We evaluated the analyses on ten Modula-3 programs ranging in size from 400 to 29,000 lines of non-blank, non-comment lines of code (76,122 total lines).

Our results demonstrate that these simple algorithms are very effective at resolving statically *monomorphic* method invocation sites, *i.e.*, those method invocation sites that call a single procedure. Our techniques detect 92% of the method invocation sites in our test suite that call one procedure at run time.

Most method invocations are resolved by type-hierarchy analysis, but intraprocedural and interprocedural type propagation, and aggregate analysis, also benefit individual programs. Our cause analysis indicates that the primary reason for analysis failure is *polymorphism*, *i.e.*, the method invocations call more than one procedure at run time and thus analysis alone cannot resolve them. Our results show that most polymorphic method invocations are due to use of heap allocated structures. For monomorphic sites, the primary cause of analysis failure is the loss of information at control-flow merges, records, and heap allocated structures. Although there is room to improve the aggregate analysis, our simple analysis resolves most of the few monomorphic calls that could be resolved by an aggregate analysis. Our analyses are simple, fast, and effective enough to incorporate into existing compilers for statically-typed object-oriented languages.

The remainder of this paper is organized as follows. Sections 2 and 3 review background material and further motivate these analyses. Section 4 describes type hierarchy analysis, type propagation, and aggregate analysis. Section 5 presents static and dynamic measurements of method invocations and our ability to resolve them. Section 6 gives the reasons for our failures and successes and compares our analyses to an oracle. Section 7 discusses how our results apply to languages like C++. Section 8 reviews related work. Section 9 concludes.

2 Background: Polymorphism through subtyping

Statically-typed object-oriented languages support polymorphism through subtyping. A type `S` is a subtype of `T` if it supports all the behavior of `T`. Thus, the program can use an object of type `S` whenever an object of type `T` is expected. In particular, a variable with *declared type* `T` may refer to objects that are subtypes of `T`, not just `T`.

Consider the Modula-3 type hierarchy in Figure 1, which defines a type `T`, and `S`, a subtype of `T`. Procedures `mT`, `mS`, and `nS` are defined elsewhere. `S` has all the behavior of `T` (in particular, the `m` method) but it may have different implementations of `T`'s methods (in this case, `mS` instead of `mT`). `S` may support behavior not supported by `T` (in this example, the `n` method). Invoking the `m` method on a variable with declared type `T` may invoke one of three procedures:

1. `mT`, if the last object assigned to the variable had type `T`;
2. `mS`, if the last object assigned had type `S`; and
3. `error`, if the last object assigned was `NULL`.

In general, invoking a method on a variable (the *receiver*) can call any procedure that overrides that

```
TYPE T = OBJECT
           f: T;
         METHODS
           m := mT;
         END;
(* S is a subtype of T *)
TYPE S = T OBJECT
         METHODS
           n := nS;
         OVERRIDES
           m := mS;
         END;
```

Figure 1: A Modula-3 Type Hierarchy

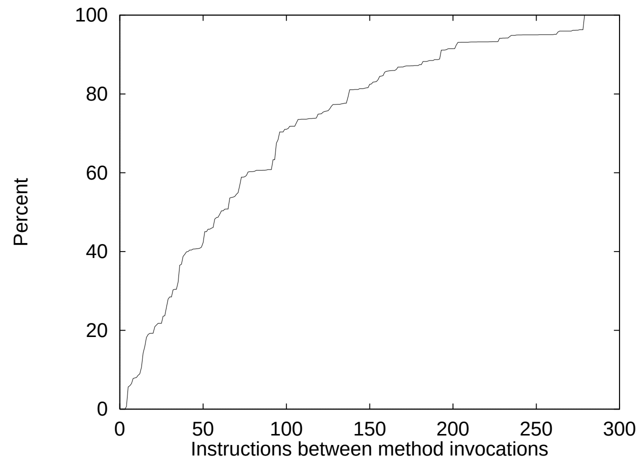

Figure 2: Instruction distribution between method invocations

method in the variable's declared type or any subtype of its declared type. The `NULL` type is a subtype of all objects in Modula-3 and overrides all methods with an `error` procedure.

A *polymorphic* method invocation site calls more than one user procedure at run time. For example, consider invoking the `print` method on each element of a linked list in a loop. If the list links objects of different types, then the `print` method invokes different procedures depending on the type of the list element.

A *monomorphic* method invocation site always invokes the same user procedure (or `error`). It may have the potential to be polymorphic but the polymorphism is never present at run time. To continue the linked list example, if the list links objects of only one type, then the `print` method will always invoke the same procedure.

A method invocation is *resolved* if it is identified as being monomorphic. The goal of the analyses presented here is to resolve all the monomorphic sites.

3 Motivation

Object-orientation promises many important software engineering benefits. There are however significant obstacles to obtaining peak performance from object-oriented programs on modern processors. In particular, small methods with frequent method invocations hinder compiler optimizations since the compiler does not know where control will go next.

Modern hardware, on the other hand, requires a substantial amount of control-flow information to exploit the hardware fully. For example, the memory latency of some modern machines is around 70–100 cycles [11] and thus the compiler needs to know which instruction will be executed 70 cycles in advance in order to prefetch data or instructions most effectively. Other hardware features that require extensive control-flow information to optimize for include multiple instruction issue per cycle and non-blocking loads.

Figure 2 shows the cumulative distribution of the number of SPARC instructions executed between method invocations for a run of `M2toM3`, a Modula-3 program that converts Modula-2 programs to Modula-3. At point `(x,y)`, `y`% of the time there are fewer than `x` instructions between method invocations. The graph shows that more than 50% of the time there are fewer than 60 instructions between method invocations. These instructions include those for passing parameters and for method lookup. With such frequent unknown control transfers, it is unlikely that the compiler can fully exploit prefetch instructions or instruction level parallelism.

There are two ways of improving the control-flow information in the compiler. First, program analysis may reduce the set of possible procedures called at each method invocation [1, 10, 12]. This analysis is effective only on monomorphic method invocations. Second, a program may be transformed so that the performance-critical method invocations can be converted to direct calls. An example is splitting, which duplicates code in order to improve type information [8]. Program transformations are effective on polymorphic and monomorphic method invocations.

Since transformations may degrade performance, a good strategy for a compiled language is first to analyze, and only then to apply transformations to convert the remaining, frequently executed, polymorphic method invocations. In this paper, we focus on the first part of the solution: to develop and evaluate a range of analysis techniques to resolve method invocations.

4 Analysis

In this section we describe each analysis technique and introduce cause analysis. We use the type hierarchy of Figure 1 to illustrate the strengths and limitations of the analyses. We use a power set of the types for the lattice for our analyses; the initial type for a variable or heap or record field or pointer reference is the empty set.

4.1 Type Hierarchy Analysis

Type hierarchy analysis bounds the set of procedures a method invocation may call by examining the type hierarchy declarations for method overrides. For each type `T` and each method `m` in `T`, type hierarchy analysis finds all overrides of `m` in the type hierarchy rooted at `T`. These overrides are the procedures that may be called when `m` is invoked on a variable of type `T`. Since `NULL` is a subtype of all objects in Modula-3 and it overrides all methods, type hierarchy analysis can never narrow down the possibilities to just one; at best it determines a method is one procedure or the `error` procedure.

Type hierarchy analysis does not examine what the program actually does, just its type and method declarations. Thus, it takes time proportional to the number of types and methods in the program, $O(|\mathit{Types}| * |\mathit{Methods}|)$.

4.2 Intraprocedural Type Propagation

Intraprocedural type propagation improves the results of type hierarchy analysis by using data flow analysis to propagate types from *type events* to method invocations within a procedure. Type events create or change type information. The three distinguishing type events are allocation (v $\leftarrow$ *NEW* (t)), implicit and explicit type discrimination operators (*IsType* (v, T)), and assignment (v $\leftarrow$ u), which includes parameter bindings at calls.

For example, consider the following code:

```
1  p := NEW (S);
   IF cond THEN
2    o := NEW (T);
3    o.m ();
   ELSE
4    o := p;
5    o.m ();
   END;
6  o.m ();
```

Statement 2 contains a type event: an allocation. The allocation propagates the type `T` to `o`, and thus determines that the method invocation in statement 3 calls procedure `mT`. Similarly, statement 4 contains a type event: an assignment. The type event propagates the type of `p` to `o`, and thus determines that the method invocation in statement 5 calls procedure `mS`. Finally, intraprocedural type propagation merges the types of `o` at the control merge before statement 6, yielding the type {`T`, `S`}. Thus the method invocation in statement 6 cannot be resolved to a single procedure.

Our implementation of type propagation propagates types only to scalars; it assumes the conservative worst case (the declared type) for the allocated types of record fields, object fields, array references, and pointer accesses.

We formulate intraprocedural type propagation as a data-flow problem similar to reaching definitions. We identify and propagate pairs of variables and sets of possible types for the variables. All variables initially have the empty type. A statement s with a type event generates and kills types as follows:

$$\textsc{GenType}\ (v \leftarrow NEW(t)) = \langle v, t \rangle$$

$$
\begin{aligned}
&\textsc{GenType}\ (\mathit{IsType}(v,T)) && = \langle v, \mathit{TypeOf}(v) \cap T\rangle \\
&\textsc{GenType}\ (v \leftarrow u) && = \langle v, \mathit{TypeOf}(u)\rangle \\
\\
&\textsc{KillType}(v \leftarrow \mathit{NEW}(t)) && = \langle v, \mathit{TypeOf}(v)\rangle \\
&\textsc{KillType}(v \leftarrow u) && = \langle v, \mathit{TypeOf}(v)\rangle
\end{aligned}
$$

T denotes a set of types and t denotes a single type. *TypeOf* returns the current types of a variable. *IsType* is an explicit type discrimination event which checks if v's type is in T. Type discrimination may also be implicit. In particular, for each *IsType*, there is an implicit type discrimination event for the false branch.

The data-flow equations for a statement s are similar to the equations for reaching definitions:

$$
\begin{aligned}
\mathrm{IN}(s) &= \bigcup_{p \in \mathrm{PRED}(s)} \mathrm{OUT}(p) \\
\mathrm{OUT}(s) &= \textsc{GenType}(s) \cup (\mathrm{IN}(s) - \textsc{KillType}(s))
\end{aligned}
$$

Our implementation stores the possible types of a variable as a set. Thus, the union and intersection operators are set union and set intersection respectively. This problem formulation is monotone and distributive.[1] Since Modula-3 programs are always reducible and type propagation is *rapid* [17], we use a $O(n * v)$ solution, where n is the number of statements in a procedure and v is the number of variables in the procedure and each step of the algorithm is a bit vector operation.

4.3 Aggregate Analysis

The goal of our aggregate analysis is to handle a common situation efficiently: monomorphic use of a general data structure. Consider the linked list package again. Our aggregate analysis detects when a program links objects of a single type and thus would resolve the invocation of the `print` method on the list elements.

The aggregate analysis circumvents the difficulty of analyzing records and heap allocated objects by merging all instances of an object or record type. For example,

```
v: T;
v.f := <rhs>
```

[1]With the type discrimination operations in Modula 3, a more precise, but non-monotone formulation is possible. As far as we know, no one has investigated this formulation.

propagates the types of `<rhs>` to the field `f` of all possible types of `v`. The possible types of `v` may be determined by another analysis (such as type propagation) or may be conservatively approximated as `T` and its subtypes.

This analysis discovers monomorphic uses of general data structures. However, if the program allocates two distinct linked lists, one with elements of type `S` and the other with type `T`, aggregate analysis does not recognize that each list is homogeneous. It infers the type {`T, S, NULL`} for the elements in both lists.

The type of an object-typed field always includes `NULL` since all fields in Modula-3 are initialized at allocation, and thus the first assignment to every object-typed field is always `NULL`. In order to propagate types to and from a field, aggregate analysis requires that all assignments to that field be available for analysis.

We perform aggregate analysis in a single forward pass. For each procedure, it is $O(n)$ time in the number of statements in the procedure.

4.4 Interprocedural Type Propagation

Interprocedural type propagation combines with intraprocedural type propagation to resolve more method invocations. It begins by building a call graph of the program. The call graph has an edge from a method invocation to each possible target determined by the earlier intraprocedural analysis. The algorithm operates by maintaining a work list of procedures that need to be analyzed. A procedure needs to be analyzed if new information becomes available about its parameters or about the return value of one of its callees. When interprocedural type propagation analyzes a procedure, it may put the callers and callees of the procedure on the work list and update the call graph. In particular, analysis may eliminate some call graph edges if it refines the type of a method receiver. Interprocedural type propagation maintains the work list in depth first order and terminates when the work list is empty.

Interprocedural type propagation also keeps track of which procedures are called only via method invocations (*i.e.*, not called directly). For these procedures, it eliminates `NULL` as a possible type for the

Analysis	Eliminates NULL	Complexity
Type Hierarchy	No	$O(\|Types\| * \|Methods\|)$
Intraprocedural Type Propagation	Yes	$O(\sum_p n_p * v_p)$
Aggregate Analysis	No	$O(\sum_p n_p)$
Interprocedural Type Propagation	Yes	$O(p \sum_p n_p * v_p)$

Table 1: Summary of analyses

first argument (`self`). If `self` is `NULL`, then `error` is invoked instead of this procedure.

Interprocedural type propagation propagates types only to scalars, and it assumes the most conservative type (the declared type) for all data accessed through pointer traversal. Interprocedural type propagation does not propagate side effects from calls and assigns the most conservative type for any variable changed by the call: the declared type. Variables potentially changed by a call include variables declared in outer scopes, globals, parameters passed by reference, and parameter aliases.

A distinguishing characteristic of our interprocedural analysis is that, unlike related work ([1, 22, 20]), our analysis is context insensitive. Rather than analyzing for every combination of call site and callee we merge the parameter types of all call sites of a procedure, and the return types of all callees at a call site. This simplification yields a much faster analysis (quadratic instead of exponential) but at the cost of some accuracy. Consider the following code:

```
PROCEDURE Caller1 () =
  t := P (NEW (T));
  t.m ();

PROCEDURE Caller2 () =
  t := P (NEW (U);
  t.m ();

PROCEDURE P (o: T): T =
  RETURN o;
```

A context-sensitive analysis would analyze `P` separately for each of its call sites and thus determine that the method invocation in `Caller1` will call `mT` and that in `Caller2` will call `mU`. Our context-insensitive analysis instead merges the parameter types for each caller of `P` and thus does not resolve the method invocations in `Caller1` and `Caller2`. We show in Section 6 that for our benchmark suite, the loss in precision is not significant.

If interprocedural type propagation is invoked on an incomplete program, it makes conservative assumptions about the parameters of procedures that could be called from unavailable code, and about return values of unavailable procedures.

Since interprocedural type propagation may analyze each procedure multiple times (in particular, recursive or potentially recursive procedures), it may be substantially slower than intraprocedural type propagation. Since information flows forward through parameters and backwards from return values, the worst case complexity is $O(|\text{procedures}| * \sum_p^{procedures} n_p * v_p)$, where n is the number of statements in procedure p and v_p the number of variables in procedure p. In practice, however, we have found it to be linear in the number of instructions, analyzing each procedure an average of 2 to 4 times.

4.5 Analyzing Incomplete Programs

An implicit assumption in the analyses described above is that the entire program (except for library code) is available for analysis. Moreover, it is assumed that the library code does not create subtypes of any types declared outside the library[2]. While we have extended our analyses to work with incomplete information, the results presented here assume the entire program (except libraries) is available. In future work, we will evaluate the effectiveness of the analyses on incomplete programs.

[2]Libraries may introduce subtypes of types declared outside the library if structural equality is used for types.

4.6 Ordering the analyses

The ordering of our analyses can make a difference in the effectiveness. For instance, if type propagation occurs only before aggregate analyses, then type propagation cannot propagate the information exposed by aggregate analysis. We have not explored these interactions between analyses experimentally; instead we have chosen the following fixed ordering.

1. type hierarchy analysis
2. intraprocedural type propagation
3. interprocedural type propagation
4. aggregate analysis
5. interprocedural and intraprocedural type propagation where needed

4.7 Summary

Table 1 summarizes the analyses. *Eliminates* `NULL` indicates whether the analysis can eliminate `NULL` as a possible type. In the *Complexity* column n_p is the number of statements in procedure p. These algorithms are simple and therefore fast, as shown in the *Complexity* column.

5 Results

Section 5.1 describes the benchmark programs. Section 5.2 evaluates the effectiveness of our analyses in converting method invocations to direct calls. Section 5.3 presents the run-time improvements due to resolving these method invocations.

5.1 Benchmark Programs

For each of the benchmark programs, Table 2 gives the number of non-comment, non-blank lines of code, the number of method invocations at compile time and at run time (for one run of the benchmark), and a brief description of the programs.

5.2 Converting Method Invocations to Direct Calls

Figures 3 through 12 illustrate the percent of method invocations resolved by each analysis for each of the benchmark programs. The graphs have one bar for each level of analysis:

tha: type hierarchy analysis

tha+tpa: tha plus intraprocedural type propagation

tha+tpa+h: tha+tpa plus aggregate analysis

tha+tpa+ip: tha plus interprocedural type propagation

tha+tpa+ip+h: tha+tpa+ip plus aggregate analysis

The black regions in the bars corresponds to percentage of method invocations at run time that the analyses resolves to exactly one procedure. The gray region corresponds to method invocations that analysis resolves to one user procedure or `error`. The pair above the bar is the corresponding number of static call sites. If a method invocation site is not executed at run time, it does not appear in these graphs.

The figures illustrate that type hierarchy analysis and aggregate analysis are most effective analyses for these programs and the effectiveness of the other analyses is relatively small. Intraprocedural and interprocedural type propagation removes many `NULL` possibilities but resolves few additional method invocations by themselves. Thus type propagation is useful for languages that have well defined semantics for the `NULL` case (such as Modula-3 and Java) but is less useful for other languages (such as C++). Type propagation will also be more effective if the whole program is not available since type hierarchy and aggregate analysis will become much less effective because of the incomplete type hierarchy.

Aggregate analysis along with type propagation resolves two method invocation sites in `dom` and 341 sites in `m3cg` (of which 88 are executed in the benchmark run). Aggregate analysis is also effective on `trestle`, resolving five method invocation sites, and on `postcard`, resolving 22 method invocation sites. However the resolved method invocations in `trestle` and `postcard` are not executed in the

Name	Lines	Method invocations Compile time	Run time	Description
format	395	37	47,064	Text formatter
dformat	602	95	30,775	Text formatter
k-tree	726	13	714,619	Builds and traverses a tree structure
slisp	1645	223	67,253	Small lisp interpreter
pp	2328	24	458	A pretty printer for Modula-3 programs
dom	6186	222	12,377	A system for building distributed applications [18]
postcard	8214	293	3,076	A graphical mail reader
m2tom3	10574	1821	19,886,862	Converts Modula-2 code to Modula-3
m3cg	16475	1808	32,850	M3 v. 3.5.1 code generator + extensions
trestle	28977	430	10,756	Window system + small application
Total	**76,122**	**4966**		

Table 2: Benchmark Programs

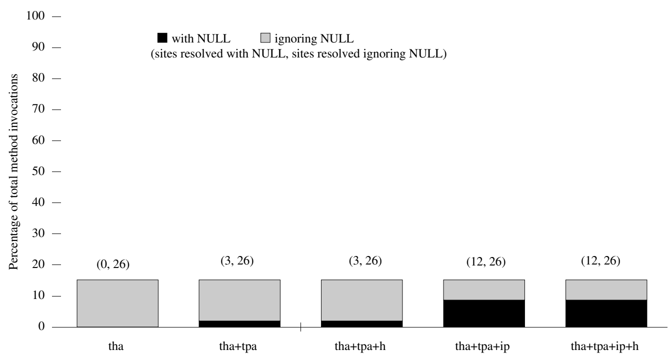

Figure 3: Analysis results for `format`

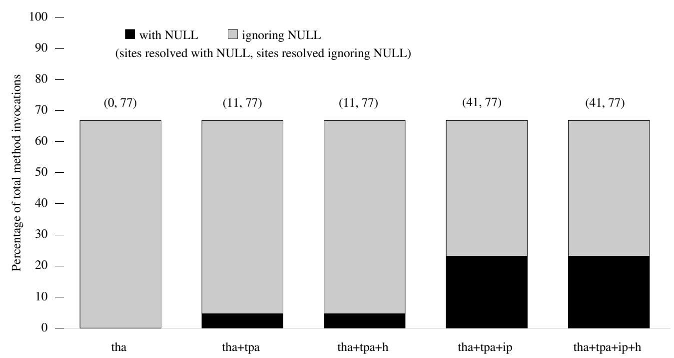

Figure 4: Analysis results for `dformat`

benchmark run and thus the impact of aggregate analysis does not appear in the figures. (`Trestle` and `postcard` are large systems and our inputs exercised only a part of them.)

5.3 Execution time improvement

To judge the run-time impact of the analyses, we ran our non-interactive benchmarks[3] before and after resolution of method invocations on a DEC 3000/400 workstation. In the first experiment, the compiler replaced method invocations that resolved to exactly one user procedure with direct calls. These are the method invocations that make up the black region in Figures 3 through 12. The compiler did not convert method invocations that resolved to one user procedure or `error` since that would be inconsistent with Modula-3 language semantics. We

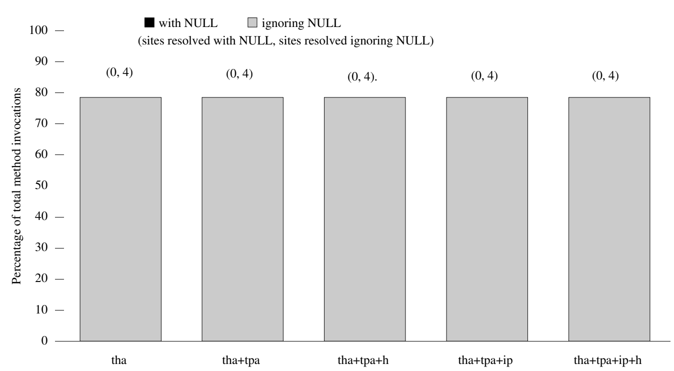

Figure 5: Analysis results for `k-tree`

[3]Because `Trestle`, `postcard`, and `dom` are interactive, we did not include them in this experiment.

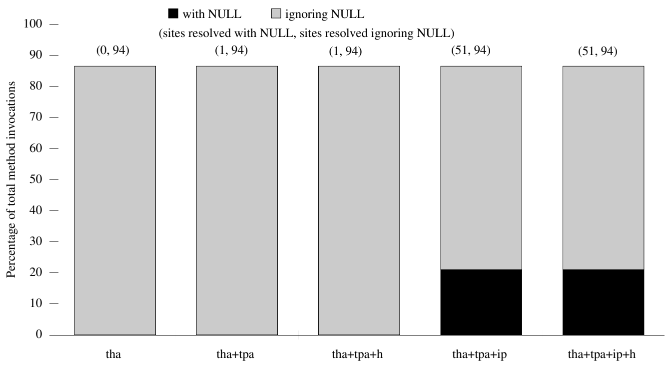

Figure 6: Analysis results for `slisp`

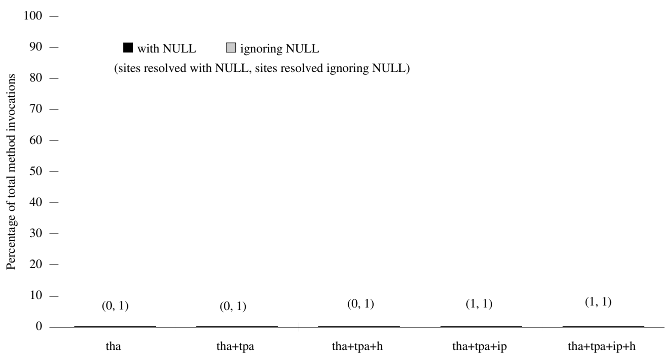

Figure 7: Analysis results for `pp`

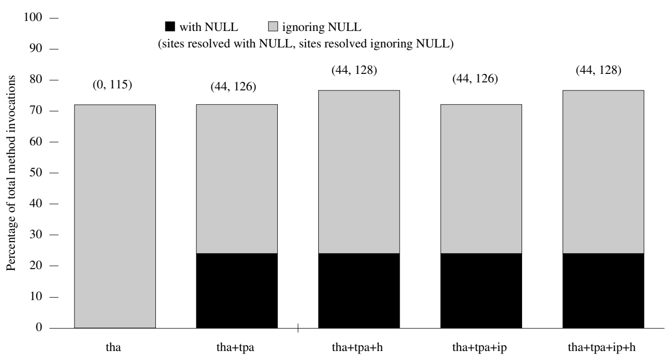

Figure 8: Analysis results for `dom`

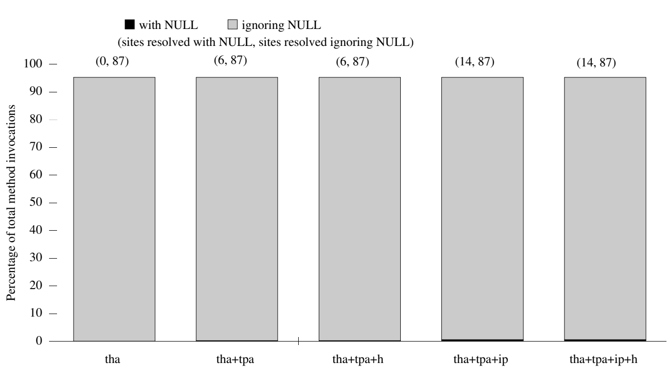

Figure 9: Analysis results for `postcard`

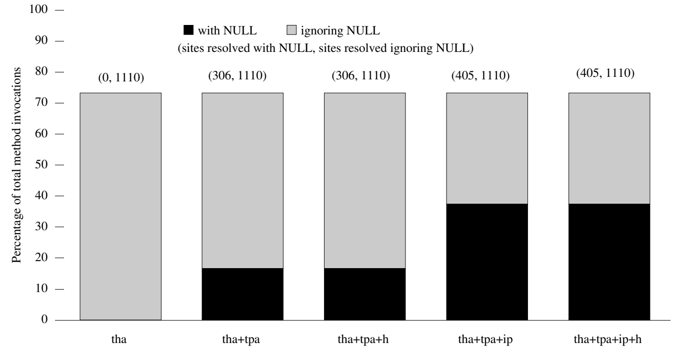

Figure 10: Analysis results for `M2toM3`

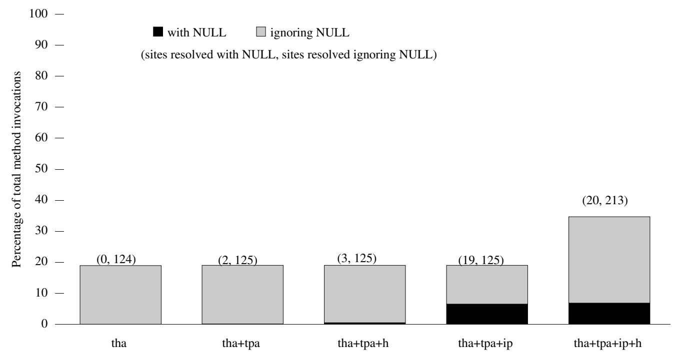

Figure 11: Analysis results for `m3cg`

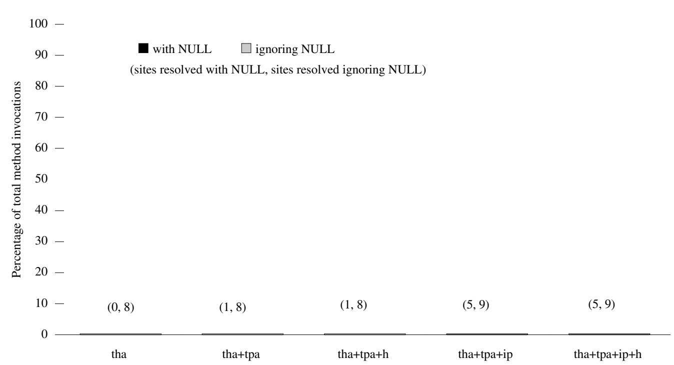

Figure 12: Analysis results for `Trestle`

found that the execution time improvement averaged less than 2% for the benchmarks even when the compiler inlined the frequently executed resolved method invocations.

In the second experiment, the compiler replaced method invocations that resolved to one user procedure or `error` with direct calls. Ignoring the `error` possibility is inconsistent with Modula-3 semantics but it facilitates comparison with languages such as C++. We found that resolving the method invocations improved performance by 0 to 9%, with an average improvement of 3%. When the compiler inlined the frequently executed resolved method invocations, the performance improvement ranged from 0 to 19%, with an average of 6.5%.

These results show that unlike dynamically-typed languages, the direct cost of method invocations in statically typed-languages is small. The main cost of method invocations is indirect: method invocations obscure control flow and thus inhibit compiler optimizations. We are currently implementing and evaluating several optimizations that exploit the information exposed by the analyses described here.

6 Cause Analysis

In Section 6.1 we describe our cause analysis technique and in Section 6.2 we apply it to the benchmark programs.

6.1 Technique

In the absence of control and data merges, such as calls, analysis could determine the allocated type of every variable. However, real programs introduce potential polymorphism by merging control and data as follows:

- Control merges:
  - after a conditional statement
  - at a call site with multiple targets (because of the returns)
  - at a procedure with multiple callers
  - at the return of a procedure with multiple return statements
- Data merges:
  - at assignments through potential aliases (includes heap allocated data, pointers, and array references)

Source	Solution
Record field	More powerful aggregate analysis
Object field	More powerful aggregate analysis
Control merge	Context sensitive analysis
Unavailable	Analyze libraries

Table 3: Cause of information loss

If a merge results in the loss of type information and the affected variable is later used to invoke a method, then that merge is the reason analysis failed to resolve the method invocation. The method invocation may actually be polymorphic, or the analysis may not be powerful enough to resolve it. For every method invocation that our analyses do not resolve, our cause assignment algorithm finds the merges that result in the loss of type information for the receiver of the method invocation. The analyzer finds the merge by following *use-def* chains [2] to the point where information is lost.

We use this information to expose the reason when our analyses fail. The reason suggests what analyses or transformations may be effective on the unresolved method invocations. For example, if a control merge obscures a type, a context sensitive analysis may prevent this loss of information. The cause analysis identifies four sources of information loss:

- **Record**: a merge of types in record fields or arrays (recall that the implementation propagates types only to scalars and to some extent to object fields),
- **Heap**: a merge of types in the heap (includes object fields and pointer references),
- **Control Merge**: a merge of types due to a control merge,
- **Code Unavailable**: a conservative type assumed due to unavailability of library code.

Table 3 suggests the techniques that may prevent the loss of information for each of the four causes of information loss.

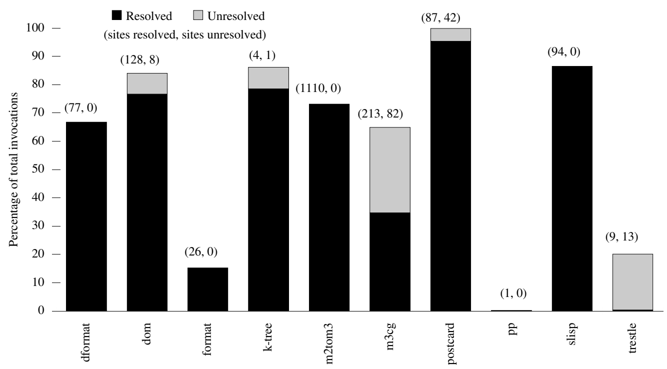

Figure 13: Monomorphic method invocations

6.2 Results

In Section 5.2, we demonstrated that the analyses resolve many method invocations to direct calls. In this section, we address these questions:

1. How do our analyses compare to an "*oracle*" that resolves *all* monomorphic method invocations?

2. What transformations will be effective in converting the polymorphic method invocations to direct calls?

Figure 13 addresses the first question. Each bar gives the run-time data for one benchmark program. The height of a bar corresponds to the percentage of method invocations that always call the same procedure in a run of the benchmark[4]. Each bar has two regions, the black region corresponds to the method invocations resolved by analysis and the gray region corresponds to the unresolved monomorphic method invocations. The pair above each bar gives the number of static method invocations corresponding to the two regions. The gray region is an upper bound on the truly monomorphic method invocations; (*i.e.*, across all possible runs of the programs) and thus on how much better the oracle can do compared to our analyses. It is an upper bound since method invocations may be polymorphic on a different program execution.

Figure 13 shows that, for all benchmarks except `m3cg` and `trestle`, our analysis resolves the vast majority of monomorphic method invocations; the analyses perform almost as well as the oracle. For `dformat`, `format`, `m2tom3`, `pp`, and `slisp`, our analyses perform as well as the oracle. Across all the benchmarks, the oracle would resolve at most 7% more method invocation sites compared to our analyses. For the benchmarks where our analyses are less effective, Figure 14 indicates which analyses may be successful in resolving these method invocations.

Each bar in Figure 14 breaks down an *unresolved* region in Figure 13 into four regions, one for each cause of analysis failure. The numbers above each bar give the total number of monomorphic method invocation sites.

For `m3cg` the figure indicates that a more powerful aggregate analysis may be successful in resolving more method invocations. On inspection of the source code of `m3cg` we found that an analysis would have to discover the semantics of a stack in order to do better than our aggregate analysis. It is unlikely that any analysis would be able to discover the semantics of a stack and thus resolve more method invocations.

For `trestle`, the primary cause of analysis failure is control merges. Thus a context sensitive analysis may be effective in resolving more method invocations. `Trestle` is the only benchmark where a context-sensitive analysis may be helpful.

Figure 15 addresses the second question: what transformations will be effective in converting the polymorphic method invocations to direct calls? Figure 15 presents data for the method invocation sites that call more than one procedure in a run of the benchmark and thus cannot be resolved by analysis alone. These method invocations are a lower bound on the polymorphic method invocations since in another run of the benchmark, additional method invocations may be polymorphic.

Figure 15 illustrates that most run-time polymorphic method invocations arise because more than one type of object is stored in a heap slot. Two techniques, explicit type test [5, 15] and cloning combined with aggressive aggregate analysis, may be able to resolve these method invocations. Merges in control are another important cause of the run-time polymorphism, especially for `trestle`, and can be resolved by code splitting and cloning [7, 14, 9].

[4] We used two runs for `pp` with different command-line parameters to expose the polymorphic method invocations.

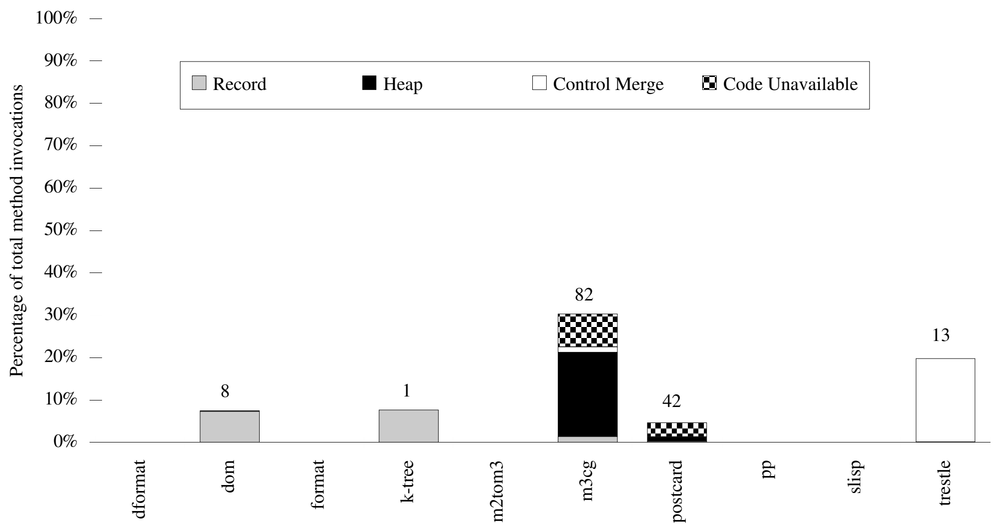

Figure 14: Monomorphic method invocations that are unresolved

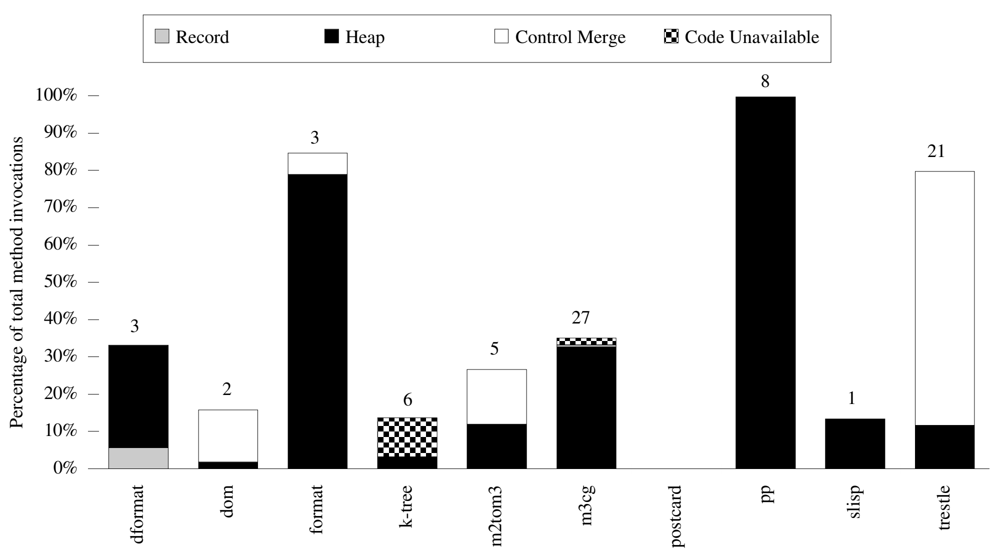

Figure 15: Polymorphic method invocations

From the static counts above the bars, we see that while the number of run-time polymorphic sites in the benchmarks is usually small, they are executed relatively frequently. For example, of the 29 method invocation sites executed in a run of `format`, only 3 sites are polymorphic, but they comprise more than 80% of the total method invocations executed. Across all the benchmarks, polymorphic sites are called 26 times more than monomorphic sites. Thus these Modula-3 programs have relatively few polymorphic method invocation sites, but they are executed very frequently. This observation has an implication for optimizations: the number of method invocation sites where transformation is needed is small and thus the code growth induced by transformations such as cloning is likely to be negligible.

7 Applicability to Other Languages

The analyses described here are language independent but their usefulness depends on the language and the programming style. For example, some C++ programming styles discourage the use of virtual functions unless necessary[5]; in essence the style encourages the programmer to attempt type-hierarchy analysis manually. In such situations, the impact of type-hierarchy analysis will be limited compared to Modula-3 programs, where all methods are virtual. We expect that our results will carry over to other statically typed object oriented languages such as C++ *if* the programs are written using only virtual methods. However the run-time improvement due to our analyses in C++ programs may be greater since method invocations are more costly in languages that have multiple inheritance. Since dynamically-typed languages encourage a fundamentally different style of programming, we expect that our results will not directly apply to them.

8 Related Work

In this section, we describe the related work on understanding and analyzing object-oriented programs and distinguish our contributions.

Fernandez [12] and Dean *et al.* [10] evaluate *type hierarchy analysis* for Modula-3 and Cecil respectively. They find that type hierarchy analysis is a worthwhile technique that resolves many method invocations. Our work confirms these results. In addition to type hierarchy analysis, we evaluate a range of other techniques.

Palsberg and Schwartzbach [20], Agesen and Hölzle [1], and Plevyak and Chien [22] describe *type inference*[6] for dynamically typed object-oriented languages. Agesen and Hölzle's, and Plevyak and Chien's analyses are more powerful than ours since they are context sensitive (polyvariant). They are also more complex and expensive. Polyvariant analyses can be used in conjunction with transformations to resolve polymorphic method invocations. We focus solely on analysis here. Plevyak and Chien discuss reasons for loss of type information, but do not present any results. We present detailed data giving reasons for loss of type information.

In work done concurrently with ours, Bacon and Sweeney [4] and Aigner and Hölzle [3] evaluate techniques for resolving method invocations in C++ programs. Bacon and Sweeney evaluate three fast analyses, including type hierarchy analysis, for resolving method invocations in C++ programs. Unlike us, Bacon and Sweeney evaluate only flow insensitive analyses. Aigner and Hölzle evaluate type feedback and type hierarchy analysis and find that they are both effective at resolving method invocations.

Pande and Ryder [21] describe a pointer analysis algorithm for C++ programs. Plevyak and Chien's type inference algorithm also does some pointer analysis [22]. Both algorithms consider the control flow in a program and are thus more powerful than our simple aggregate analysis, which also deals with pointer analysis. However, they are also much slower than our aggregate analysis. On a SPARC-10, Pande and Ryder's algorithm takes as much as 23 minutes to analyze programs that are less than 1000 lines of code (median 36 seconds). Our aggregate analysis takes 38 seconds to analyze 28,977 lines of code on a DEC 3000/400. We show

[5] Only virtual functions may be overridden in subtypes.

[6] "Type propagation" and "type inference" are terms that have been used to describe the same kinds of analysis in object-oriented languages.

that our simple analysis is effective and there is little to be gained by a more powerful analysis for our benchmarks. This result is partly due to Modula-3's language semantics which restrict aliasing; a more powerful alias analysis may be more useful for C++ than for Modula-3, but this need has not yet been demonstrated.

Chambers [6], Calder and Grunwald [5], Hölzle and Ungar [15], and Grove *et al.* [13] describe transformations for converting method invocations to direct calls. We focus solely on analysis here.

Shivers [23] describes and classifies a range of analyses to discover control flow in Scheme programs. Our interprocedural type propagation is similar to his 0CFA. While Shivers focuses on powerful (and slow) analyses—0CFA is the least powerful analysis he considers—we focus on simple and fast analyses. Interprocedural type propagation is the most complicated analysis we consider.

Another key difference between our work and that of others is that we present results that give the reason when analysis fails, and place upper bounds on how well more powerful analyses or transformations can possibly do.

9 Conclusions

We describe and evaluate a range of analyses for object-oriented programs: *type-hierarchy analysis*, *intraprocedural and interprocedural type propagation*, and *aggregate analysis*. Aggregate analysis is a new technique and is simpler and faster than previous work.

We demonstrate that our techniques are extremely effective at resolving method invocations in Modula-3 programs. On average, our analyses resolve more than 92% of the method invocation sites that are amenable to analysis and improve the run-time of the benchmark programs by up to 19%.

For method invocations that are unresolved by our analyses, we determine the reason for analysis failure. The failure reason suggests which other analyses and transformations may be effective. The primary failure reason in our benchmarks is polymorphism: the method invocations called more than one procedure at run time and thus are not amenable to analysis alone. Most of this polymorphism is due to objects of different types being stored in heap slots. The other significant reasons for analysis failure are an insufficiently powerful aggregate analysis and lack of a context sensitive analysis. Improving the aggregate analysis and adding a context sensitive analysis would resolve at most 7% more method invocation sites.

10 Acknowledgments

We would like to thank Ole Agesen, Darko Stefanovic, and the anonymous referees for comments on drafts of this paper.

References

[1] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–107, Austin, Texas, October 1995. ACM.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *European conference on object-oriented programming*, Linz, Austria, July 1996.

[4] David Bacon and Peter Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA'96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, San Jose, CA, October 1996. ACM, ACM Press.

[5] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, January 1994.

[6] Craig Chambers. *The design and evaluation of the SELF compiler, an optimizing compiler for object-oriented programming languages.* PhD thesis, Stanford University, CA, March 1992.

[7] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, Oregon, June 1989. *ACM SIGPLAN Notices 24*, 7 (July 1989).

[8] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN '90 Conference on*

Programming Language Design and Implementation, pages 150–164, White Plains, New York, June 1990. *ACM SIGPLAN Notices 25*, 6 (June 1990).

[9] Craig Chambers and David Ungar. Making pure object oriented languages practical. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, Phoenix, Arizona, October 1991. *ACM SIGPLAN Notices 26*, 11 (November 1991).

[10] Jeffery Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[11] Digital Equipment Corporation. *DEC3000 300/400/500/600/800 Models: System Programmer's Manual*, first printing edition, September 1993.

[12] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995. SIGPLAN, ACM Press.

[13] David Grove, Jeffery Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Austin, Texas, October 1995. ACM.

[14] Mary Wolcott Hall. *Managing Interprocedural Optimizations*. PhD thesis, Rice University, Houston, Texas, April 1991.

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

[16] Bill Kalsow and Eric Muller. *SRC Modula-3 Version 3.5*. Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, 1995.

[17] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 7(3):305–318, 1976.

[18] Farshad Nayeri, Benjamin Hurwitz, and Frank Manola. Generalizing dispatching in a distributed object system. In *Proceedings of European Conference on Object-Oriented Programming*, Bologna, Italy, July 1994.

[19] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, New Jersey, 1991.

[20] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–162, Pheonix, Arizona, October 1991. SIGPLAN, ACM Press.

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

[22] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of conference on object-oriented programming systems, languages, and applications*, pages 324–340. ACM, October 1994.

[23] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1991.