

# Optimizing Object-Oriented Languages

Craig Chambers

Summary by Jonathan Aldrich

May 30, 2001

## 1 Introduction

Object-oriented and functional programming languages pose new challenges for program analysis because control flow can depend on data values. For example, object-oriented languages provide dynamic dispatching. An expression of the form `receiver.msg(args)` may invoke one of several different method bodies, depending on the run-time class of the `receiver` object. In functional languages, a function call through a variable of function type may invoke any function that is possibly stored in the variable. Function pointers in C are another form of data-dependant call. These data-dependant function calls can degrade performance in functional and object-oriented languages. The performance degradation is due both to the direct cost of implementing the call and the indirect costs. Optimizations such as static binding, inlining, and procedure specialization can reduce these costs, but can only be effectively applied if an analysis can determine precisely which methods might be invoked at a given call site.

### 1.1 Call graph construction

Call graph construction is a kind of program analysis that determines which functions or methods might be invoked at each call site in the program. Shivers [S88] called this control-flow analysis. Another name is closure analysis in functional languages; it determines which closures each expression could refer to, thus implicitly determining a call graph. In object-oriented languages, the call graph is constructed using class analysis, an analysis that determines which classes an expression might belong to.

Call graph construction is challenging in part because of a circular dependency between control flow and data flow. In higher-order (functional and object-oriented) languages, the function invoked at a call site may depend on the data that flows to the call site. However, in order to determine the data flow in a program, a call graph must be built. Thus, call graph construction in higher-order languages must simultaneously build the call graph and propagate analysis data through the program.

In the rest of this paper, we will focus on object-oriented examples, but the same techniques can be used to analyse functional programs.

### 1.2 Class Analysis

Class analysis computes for each object-valued expression the set of classes that the object might be an instance of. Thus the domain of class analysis is the powerset of the set of all classes in a program, and the partial order is given by the subset operator (assuming abstract interpretation conventions). Note that in order to represent this set explicitly, the whole program must be available. For example, analysis may determine that a receiver object is an instance of one of the classes  $\{ C_1, C_2, \dots, C_n \}$ . In general, if a class  $C_i$  is in an analysis set, subclasses of  $C_i$  are not automatically in the set. The analysis is sound if, for all possible executions of the program, the run-time receiver object is an instance of one of the computed classes.

Class analysis can be performed intraprocedurally, by flowing class sets from one expression to another. The analysis must assume that all procedure arguments and results of called procedures could have a class type that is any subtype of the declared static type. At `new` statements the analysis can narrow the class sets to a single class, and if a class test is used in an `if` statement (e.g., `if (obj instanceof Class)` in Java), the class sets can be narrowed appropriately in the branches of the `if`. The results of class analysis can be used to inline or statically bind a method call site, and can even be used to constant-

fold class tests. Due to the conservative assumptions about method arguments and results, however, intraprocedural class analysis is usually not very precise, and so interprocedural class analysis must be applied.

### 1.3 Comparison to other analyses

It is interesting to compare class analysis to similar value analyses. Constant propagation computes the actual value of a constant, not just its class, and is more precise in this way. However, class analysis is more precise in that it can compute an arbitrary set of classes, while constant propagation either computes the exact value of a constant or nothing at all. Class analysis is less precise than points-to analysis, because the latter can compute that a reference points to a particular piece of data, rather than a general class of data.

This shows that design of the analysis domain is a crucial decision in program analysis. The design of the domain often determines the precision of the resulting analysis. One could improve the precision of class analysis, for example, by adding a set of constant values to the domain below each singleton class set. This would enable class analysis to perform constant propagation as well.

## 2 Previous Work

The earliest class analyses were intraprocedural flow-sensitive analyses defined in Typed Smalltalk [J88] and Self [CU90]. In that year, Shivers defined the 0-CFA interprocedural context-insensitive flow-insensitive control-flow analysis for Scheme [S88]. This analysis computes the set of closures that could be present in each program variable; when the set for a function argument is augmented, the function is reanalyzed with respect to the new data. Palsberg and Schwartzbach applied a similar algorithm to object-oriented languages [PS91]. These analyses are all flow-insensitive as defined, but some flow sensitivity can be regained by using an SSA intermediate form. The results of class tests can be used to further refine these analysis in a flow-sensitive way.

Shivers' k-CFA family of analyses extends the 0-CFA analysis to be context-sensitive [S88]. In these analyses, a function is analyzed separately for each calling context, where a calling context is a call string of k enclosing functions. These analyses have also been applied to object-oriented languages [OPS92]. Plevyak and Chien defined an adaptive version of k-CFA that begins with 0-CFA and iteratively refines the analysis by increasing k at particular program points to avoid loss of precision [PC94]. Agesen defined an analysis using the functional approach to interprocedural analysis [SP81]. His Cartesian Product Analysis (CPA) analyzes each function once for each unique tuple of argument classes [A95]. The analysis function is then encoded with partial transfer functions.

One might think that the functional approach is always superior to call strings for the following reason: Call strings may not do enough work, because a particular bound on the call-string depth may not be enough to capture the difference between different contexts. At the same time, call-strings often wastes considerable analysis effort by reanalyzing a procedure multiple times with essentially the same analysis information. In contrast, the functional approach always does exactly the right amount of work to distinguish between different calls to the same function, given a particular choice of an analysis domain.

The call strings approach does have one advantage, however: there's a natural way to bound call strings at k levels deep. It's much harder to bound the functional approach when the domain is infinitely wide (or just too large). One can limit analysis to k partial transfer functions, but that is more arbitrary than k levels of call-string context because it means that the order of analysis matters.

Another area of precision is creation-point sensitivity: does the analysis distinguish between objects created at different points in the program source? The advantage of creation-point sensitivity is that the analysis can detect when generic data structures are used in a parameterized way. For example, the analysis can distinguish a list of integers from a list of reals if they are created at different program points, and avoid assuming that an integer could be read out of the list of reals. A limitation is that different creation points may have the same parameterization, but the analysis will waste work distinguishing them. In addition, since a "factory method" may be used to create generic data structures with different parameterizations, the

analysis may need to include an arbitrary amount of context from the creating procedure to be precise enough.

### 3 A Unifying Framework

Grove and Chambers developed a generic framework for describing call graph construction algorithms, allowing their precisions and costs to be compared both theoretically and practically [GC01][G98][GD97]. The language in the framework has first class, nested functions as well as objects, and can express all of the analyses described above.

The framework describes context sensitivity in terms of contours. A *contour* is one instance of a calling context, and can also be thought of as an analysis-time specialization of a procedure. An analysis of a function for a particular call string would be an example of a contour when using call strings for context sensitivity. Every contour has a *contour key* which describes the context to which it applies, and analysis data that has been computed for that contour. For example, in the contour described above, the call string might be the contour key, and the analysis data would include the classes of the formal parameters.

Contours are created to represent procedure contexts, class contexts, and instance variable contexts. As described above, a procedure contour might represent all invocations with a particular call string suffix, or it might represent all invocations with a particular tuple of argument classes (in the CPA algorithm). A class contour might represent all instances of a class that were created at a particular program point. An instance variable contour typically uses the same contour key as the enclosing class.

In order to perform analysis within the framework, a client must specify the domain of class keys, the domain of procedure keys, and two functions. The PKS function selects and returns a contour for a particular invocation. It is given the calling procedure contour, the call site identifier, a set of class contours for each argument position, and the name of the callee procedure. The CKS function selects and returns a contour for a particular object creation. It is given the creation site id and the contour of the creating procedure.

#### 3.1 Examples

	0-CFA	0-CFA+cp	1-CFA	k-CFA	CPA	SCS
ProcKey	{ $\perp$ }	{ $\perp$ }	CallSites	Seq <sup>k</sup> (CallSites)	Tuple(ClassContours)	Tuple(Set(ClassContours))
ClassKey	{ $\perp$ }	CreationSites	{ $\perp$ }, or CreationSites [ $\times$ ProcKey]	[similar to 1CFA]	[similar to 1CFA]	[similar to 1CFA]
Cost	$O(n^3)$	$O(n^3)$	polynomial	polynomial	polynomial	exponential

**Table 1.** Common analyses expressed in the unifying framework.

Table 1 shows how the analyses discussed above can be expressed in this framework. 0-CFA is context-insensitive; thus there is only one contour key for procedures and classes, denoted by a class key set { $\perp$ }. 0-CFA can be enhanced using the set of creation points as the class key, enabling the analysis to treat data structures polymorphically without any worst-case analysis cost. For 1-CFA, the procedure key is the call site that the procedure was called from, and k-CFA generalizes this to a sequence of call sites. As in the 0-CFA case, these analyses can be run with the class key set { $\perp$ }, or they may use the creation points as class keys, or they may use both the creation point and the procedure key of the calling function as a class key. The CPA analysis uses a tuple of class contours as the key for procedures.

A novel instance of this framework is the simple class sets algorithm (SCS). Instead of using a tuple of classes as the procedure contour key, it uses a tuple of sets of classes. This makes SCS potentially take exponential time. However, simple class sets works better than CPA in practice, because when a function is called with a number of different classes in each argument position, SCS will just analyze the function once, while CPA will analyze it once for every tuple of classes in the cartesian product of the argument class sets.

## 3.2 Linear-time analyses

Unfortunately, none of the interprocedural algorithms described above scale to large programs. Recently, a good deal of work has gone into developing algorithms that compute relatively precise results, yet run in near linear time. The simplest such analysis is Class Hierarchy Analysis (CHA), which determines the set of classes in the program, and assumes that each expression may be an instance of the set of classes that inherit from its static type [DGC95]. Rapid Type Analysis (RTA) analysis improves on CHA by computing the set of classes that may be instantiated at run time [BS96]. Classes that are never created, or whose only creation sites are statically unreachable, may not be part of any class sets. Steensgaard's alias analysis [S96] shows how to compute better alias information than is attainable through RTA using unification techniques. Steensgaard's analysis can be applied to object-oriented programs as well, producing a linear time bound if an intermediate dispatching function is placed between callers and callees of a particular method name (to eliminate the possibility of  $O(n^2)$  edges in the call graph).

## 3.3 Blending Unification with Propagation

A new set of algorithms was developed that blends unification with propagation to produce a linear-time analysis that is still reasonably precise in practice [DGC98]. Consider an assignment  $x := y$  in a program. In the 0-CFA analysis, this assignment can be modeled by inserting an inclusion constraint between the values of  $x$  and  $y$ , so that  $x$  may point to any class that  $y$  may point to. Steensgaard's algorithm models this assignment with an equality constraint, so that  $x$  and  $y$  must point to the same classes. The equality constraint enables Steensgaard's analysis to run in near-linear time, but loses considerable precision by ignoring the directionality of assignment.

The key idea of the new algorithms is to use an inclusion constraint with an integer bound. Whenever a data value is passed along the constraint, the bound is decremented, and when it reaches zero, the inclusion constraint becomes an equality constraint. If the bound is 0, we have Steensgaard's analysis, and an infinite bound corresponds to 0-CFA. A constant bound  $k$  improves the precision of the analysis while still remaining linear. In practice, using a constant bound achieves about half of the optimization benefit of 0-CFA at a tiny fraction of the cost.

## 4 Future Work and Conclusions

A number of precise interprocedural algorithms have been proposed for constructing call graphs in object-oriented and functional languages. We have discussed a unifying framework that allows these analyses to be compared according to precision and cost [GC01]. Recent work in the area has focused on developing linear-time analyses that scale to large programs without losing precision.

There are a number of open questions and areas for future work:

- Can one do better using a constraint-based formalism? This might enable graph optimizations such as collapsing cycles, etc., allowing more precise analyses to scale to larger programs.
- Is there a way to extend the linear-time algorithms to make them context sensitive, perhaps using polymorphic type based analysis?
- Is there a way to analyze the contents of data structures more precisely? For example, can arrays be modeled in a more detailed way than as an unordered bag of objects? Is there a way to recover the parameterization on generic data structures?

## References

- [A95] O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In Proceedings of the European Conference for Object-Oriented Programming, 1995.
- [BS96] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1996.

- [CU90] C. Chambers and D. Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In Proceedings of the Conference on Programming Language Design and Implementation, 1990.
- [DGC95] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In Proceedings of the European Conference for Object-Oriented Programming, 1995.
- [DGC98] G. DeFouw, D. Grove, and C. Chambers. Fast Interprocedural Class Analysis. In Proceedings of the Symposium on Principles of Programming Languages, 1998.
- [J88] R. E. Johnson, J. O. Graver, and L. W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1988.
- [G98] D. Grove. Effective Interprocedural Optimization of Object-Oriented Languages. Ph.D. Thesis, University of Washington, 1998.
- [GDDC97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1997.
- [GC01] D. Grove and C. Chambers. A Unifying Framework for and Assessment of Call Graph Construction Algorithms. Draft Manuscript, 2001.
- [OPS92] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making Type Inference Practical. In Proceedings of the European Conference for Object-Oriented Programming, 1992.
- [PS91] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1991.
- [S88] O. Shivers. Control-flow analysis in Scheme. In Proceedings of the Conference on Programming Language Design and Implementation, 1988.
- [S96] B. Steensgaard. Points-to Analysis in Almost Linear Time. In Proceedings of the Symposium on Principles of Programming Languages, 1996.
- [SP81] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds., Prentice-hall, Chapter 7, 189-233, 1981.