Static Analysis of Object-Oriented Programs

Christoph Steindl
steindl@ssw.uni-linz.ac.at
Institute for Practical Computer Science
Johannes Kepler University Linz

Abstract

Static analysis derives information by inspection of the source code. The information must be valid for all possible executions of the program. Conservative assumptions must be taken if the program uses conditional branches and iteration since it is not known at compile time which branches will be taken at run time and how many iterations there will be. Two main concepts of object-oriented programming are polymorphism and dynamic binding. Furthermore object-oriented programs make heavy use of pointers (or references to objects). These dynamic aspects are difficult to integrate in static analysis. We have developed a program slicer that models dynamic aspects of object-oriented programs correctly. Starting from conservative assumptions about dynamic binding and aliases, new user guidance techniques are used to reduce these assumptions.

1 Introduction

Static analysis derives information by inspection of the source code. The information must be valid for all possible executions of the program. Conservative assumptions must be taken if the program uses conditional branches and iteration since it is not known at compile time which branches will be taken at run time and how many iterations there will be. Dynamic analysis can derive information by monitoring one particular execution of the program. It can consider the actual values of the variables during this execution. Therefore, the information is only valid for the particular execution but not in general. Static information is necessarily less precise than dynamic information. On the other hand, it can be computed once for all possible executions, whereas dynamic information must be computed again and again.

Two main sources of imprecision of static analysis are dynamic types of polymorphic variables and alias definitions: In most cases the exact destination of polymorphic call sites cannot be determined by static analysis. Dynamic binding can only be handled conservatively by considering the effects of all possible call destinations at polymorphic call sites. On the other hand, data flow analysis for heap allocated objects is difficult. Since the number of objects is unbounded, they cannot be handled individually. If they are classified into groups, then all objects of a group are aliases for the data flow analysis.

We have implemented the Oberon Slicing Tool (see [OST] and [Ste99] for information about the OST, and [Ste98a, Ste98b] for a technical description), a program slicing [Wei81] tool for Oberon-2 [MöWi91]. Oberon-2 is a hybrid object-oriented language in the tradition of Pascal and Modula-2. It provides classes with single inheritance that are declared within modules. Program slicing is a program analysis and decomposition technique that extracts from programs statements that are relevant to a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the value of variable *v* at statement *s*?"

The Oberon Slicing Tool computes an internal program representation that models the relationships between the program entities by control dependences and data dependences. A program node *x* is *control dependent* on another program node *y* if it depends on *y* whether *x* is executed or not. A program node *x* is *data dependent* on another program node *y* if the value computed for a variable at node *y* may be used at node *x*. First, both kinds of dependences are computed in a conservative way. Then program slicing can be used to restrict the program to the parts that are relevant to the part of the program that is currently

investigated by the user. The user can restrict the conservative assumptions about control and data flow and initiate the recomputation of control and data flow information. In this way, static analysis can be enriched with user-supplied knowledge to yield information with a precision similar to dynamic information.

The rest of this paper describes how the OST handles the dynamic aspects of object-oriented programs: Section 2 discusses the handling of dynamic binding as part of control flow analysis. Section 3 explains the handling of aliases as part of the data flow analysis. Section 4 describes how other features of object-oriented programming can be handled by static analysis. Section 5 gives concluding remarks.

2 Dynamic Binding

In object-oriented programs, messages are sent to objects instead of calling procedures. Objects react to these messages by performing the appropriate operation. This means that there may be several implementations for the same message. The sender of the message does not know how the receiver of the message will react. Much of the power of object-oriented programming stems from polymorphism, the possibility that pointers to objects may refer to objects of different classes at run-time. Heterogeneous sets of objects can be treated uniformly at the sender side while appropriate reaction on the receiver side is guaranteed. Since the exact reaction to a message send depends on the dynamic type of the receiver, it is not known until run time. This means that it cannot be determined at compile time by static analysis.

The main problem of static control flow analysis is to derive precise approximations for the possible call destinations. Dean et al. used Class Hierarchy Analysis [DGC94], whereas Pande and Ryder [PaR93] used flow-sensitive type analysis. Zhang and Ryder showed that the determination of static types in the presence of procedure variables is NP-hard in most cases [ZhR94].

Our approach is to handle the effects of dynamic binding first conservatively by computing for each call site the set of possible call destinations. Then we present the results to the user and let the user decide if we overestimated the set of destinations. The user can enable and disable some of the call destinations. The set of enabled call destinations is then used to perform data flow analysis.

In Oberon-2, there are two possibilities for dynamically bound calls: calls of methods and calls via procedure variables. For methods, the receiver can either be monomorphic or polymorphic. If the receiver is monomorphic, it will refer to objects of only one kind at run time. This means that the objects will all react by executing the same method. If this method can be determined at compile time, the method call can be handled precisely by static analysis. If the receiver is polymorphic, it will refer to objects of different kinds. The kinds of the objects can be safely approximated, since the type system of the programming language guarantees that the dynamic type is either equal to the static type or any extension of the static type. In Oberon-2 we can apply the following rules to restrict the sets of call destinations:

1. Super calls can be bound statically since it is known at compile time which is the overridden method in the super class.
2. Methods can be bound statically if
   a) the actual receiver is not exported and there are no extensions of this record type (no overridden method possible), or
   b) the record type is not exported and there are no extensions of this record type (no overridden method possible), or
   c) the record type is not exported and the method is not overridden in any of the extensions within the same module.

The rules 2.b) and 2.c) describe the cases where the method is surely not overridden (this corresponds to an empty Override set of the method in the terms of [Bac97]). For dynamically bound method calls, fast techniques like Rapid Type Analysis [Bac97] or flow sensitive techniques [PaR93] can be used to restrict the sets of call destinations.

For procedure variables, the possible call destinations can be approximated by the set of matching procedures that are somewhere assigned to a procedure variable. In Oberon-2, two procedures only match, if they (see [MöWi91]),

a) they have the same number of parameters, and
b) they have either the same function result type or none, and
c) parameters at corresponding positions have equal types, and
d) parameters at corresponding positions are both either value or reference parameters.

The set of call destinations could be further reduced by propagating the assigned procedures along all possible paths in the invocation graph [EGH94].

Zhang and Ryder showed that the determination of static types in the presence of procedure variables is NP-hard in most cases [ZhR94]. It is therefore reasonable to use approximations in the first place, and let the user restrict the effects of dynamic binding according to the use case that he has in mind. The sets of enabled call destinations is then used to perform data flow analysis for the use case that the programmer wants to investigate. The derived information is no longer valid for all possible executions. Note that we are not using this information for improving the performance of the program but rather to assist the programmer during the interactive process of debugging and program understanding.

3 Aliases

When control flow analysis has determined the call destinations of call sites, data flow analysis can be performed. For structured programs with scalar variables, Aho et al. [ASU86] describe an efficient technique to compute reaching definitions: A program node *x* is data dependent on all program nodes *y* that are reaching definitions for the operands used at *x*. However, there are a few language constructs that complicate data flow analysis:

- Reference parameters may introduce aliases. Two variables are aliases if they refer to the same memory cell. Aliases lead to less precise data flow information.
- Structured data types like arrays and records complicate the computation of data flow information.
- Heap-allocated objects pose additional problems: Since the number of objects is unbounded, they cannot be individually. If they are classified into groups, then all objects of a group are aliases for the data flow analysis.

For strongly typed programming languages, type information may be used to restrict the sets of possible aliases. Additionally, information about the place of the declaration of the variable can be used to restrict the sets of possible aliases. Finally, feedback from the user can be used to restrict the sets of possible aliases.

Our approach is as follows: When computing the sets of possible aliases for a procedure, we start with the scope S of accessible objects. These include the local variables and parameters, as well as intermediate and global variables. When a variable $a \in S$ is defined, all other variables $b \in S$ might as well be changed. This means that the value of *b* is either affected by the definition of *a* (if *a* and *b* are aliases) or not (if *a* and *b* are not aliases). Since Oberon-2 is a statically typed programming language with strong type checking, we can use type information in order to restrict the set of possible aliases of *a*. The type system guarantees, that memory cells cannot be interpreted differently when accessed via different variables. This means that *a* and *b* may only be aliases if they have the *same type* (see [MöWi91] for details). Information about the place of the declaration of the variables can be used to further restrict the sets of possible aliases. Therefore, we first recapitulate the different possibilities to declare variables:

- Global variables are declared at the module level. They reside within the block of global data of a module. They are also called static data, since they are allocated when the module is loaded and they stay in memory until the module is unloaded. Each global variable is allocated at a different address, no two global variables may refer to the same memory cell. They are accessible from anywhere within the declaring module. Additionally, they may be exported by the declaring module and imported into other modules. This export can be either read-only or read/write (thereby granting the importing module full access to the object).
- Local variables are declared within a procedure. They are allocated for each activation of a procedure (in most implementations on the stack). Each local variable is allocated at a different address, no two local variables may refer to the same memory cell. They are also called automatic data, since they are allocated when the procedure is called and their memory is automatically reclaimed when the procedure returns. They are only accessible within the declaring procedure and procedures that are nested within the declaring procedure.
- Intermediate variables are local variables of a procedure *P* that are accessed from within a procedure *Q* that is nested in *P*. When regarding these variables from procedure *P*, they are ordinary local variables, when regarding them from procedure *Q*, they are intermediate variables, since they are neither local to *Q*, nor global, but intermediate.
- Objects and arrays may be allocated on the heap. They are referenced by and accessible via pointer variables. Two pointers may reference the same object. Heap data is also called dynamic data, since it is allocated on demand (by a NEW statement) and its memory is automatically reclaimed by the garbage collector when it is no longer referenced.

There are two kinds of parameters in Oberon-2:

- Value parameters can be considered as local variables where the values of the expressions at the call site are used as initial values of the formal parameters. Memory is automatically allocated and reclaimed as for local variables.
- Reference parameters can be considered as additional names for the actual parameters. No new memory is allocated for reference parameters. Reference parameters are the main source of aliases in Oberon-2.

With the information about the place of the declaration of a variable, one can restrict the sets of possible aliases. Table 1 contains one row and one column for local variables (including local value parameters), global read-only variables of imported modules, intermediate variables (including intermediate value parameters), global variables of the module under consideration, and global variables of imported modules that have been exported without access restrictions, and reference parameters. In each cell of the table, "no" indicates that two objects *o1* (row) and *o2* (column) may not be aliases, whereas "yes" indicates that the two objects may be aliases. The table is symmetric, i.e. *Cell(x1, y1) = Cell(y1, x1)*.

o1 \ o2	(1)	(2)	(3)	(4)	(5)	(6)
(1) local var	no	no	no	no	no	no
(2) global var (other mod.), read-only	no	no	no	no	no	no
(3) intermediate var	no	no	no	no	no	yes
(4) global var (this mod.)	no	no	no	no	no	yes
(5) global var (other mod.), r/w	no	no	no	no	no	yes
(6) reference par	no	no	yes	yes	yes	yes

Table 1 - Possible aliases

Local variables, intermediate and global variables may never be aliases since they are allocated at different addresses. Reference parameters are aliases of the actual parameters. At the call site, intermediate variables with smaller nesting level ("yes" in last column of row 3 in Table 1), global variables without access restrictions (rows 4 and 5 in Table 1), and other reference parameters with smaller nesting level (row 6 in Table 1) may be used as actual parameters.

The sets of possible aliases restricted by type information and by information about the declaration of the variables is the most precise approximation, as long as we do not regard the actual parameters at call sites. For incomplete programs, such as frameworks, we cannot further restrict the sets of possible aliases. If we analyze complete programs ("closed-world assumption"), then we could further reduce the sets of possible aliases.

Structured data types raise another problem: the actual parameter at a call site may be a part of a structured variable, e.g. a field of a record or an element of an array. The variable may be allocated statically, automatically, or dynamically. So it is not enough, to test whether two variables *a* and *b* of the scope have the same type when computing the possible aliases for the definition of *a*, but additionally whether variable *a* can be contained in *b* or whether *b* can be contained in *a*. *a* can be contained in *b* in the following cases:

- if *b* is a record of which *a* might be a field
- if *b* is an array of which *a* might be an element
- if *b* is a pointer that is dereferenced and *a* might be a field of the referenced record
- if *b* is a record and its type is an extension of the type of *a*

Automatically allocated objects (i.e. locally declared objects) are treated individually. Heap-allocated objects of a particular type are handled uniformly, meaning that they may be aliases of each other. Additionally, objects of a type *T1* derived from *T0* are considered to be aliases of objects of type *T0.*

The user interface of the Oberon Slicing Tool visualizes the sets of possible aliases. The user can disable some or all of the possible aliases. He can then initiate the re-computation of data flow information where only the enabled aliases lead to non-killing definitions.

4 Other Features of Object-Oriented Programming

Type-extension is used to inherit fields and methods. Inherited fields and methods can be accessed just like new fields and methods of a class. Data flow and alias analysis must be aware of type-extension.

Since classes are made for later re-use, static analysis must always consider possible extensions. It is not wise to analyze them under the "closed-world assumption". It is more appropriate to envision possible extensions of the class and to make the hot spots visible by highlighting places where yet unknown methods might be called.

Information hiding is one of the strengths of object-oriented programming. It abstracts from details and provides access only via the external interface. Internal data is not accessible. However, the state of objects is also defined by these internal data fields. Assume an object *o* with two methods *M1* and *M2*. *M1* performs some operation and changes the internal field *x* of *o*. *M2* reports the value of the internal field *x*. In the following piece of source code, the first method call must be included into the slice for the last statement, since it defines a value that is used by the last statement.

```
BEGIN
   o.M1;              (* changes o.x *)
   o.M2               (* outputs o.x *)
END
```

For program understanding, it is necessary to unveil some of the implementation details.

5 Conclusions

Static analysis is not immediately applicable to object-oriented programs. Techniques must be developed to account for the run-time effects of dynamic binding and aliasing. Starting from conservative assumptions about the effects of dynamic binding and aliases, user feedback can be used to restrict the dynamic effects. Data flow information can be re-computed, leading to more precise information. In combination with program slicing, only a few cycles of restricting and recomputing are necessary to model the use case that the programmer currently investigates.

Another approach would be start from optimistic assumptions about dynamic binding (disable all call destinations) and aliases (disable all possible aliases), and then execute a few test cases thereby enabling the call destinations and aliases witnessed during the test cases. The results could be visualized and presented to the user. Again, he would be able to enable/disable specific call destinations and aliases in order to model the use case that he wants to investigate.

Bibliography

[ASU86] A. Aho, R. Sethi, J. Ullman: *Compilers: Principles, Techniques, and Tools*.
Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Bac97] D. Bacon: *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*.
PhD thesis, University of California at Berkeley,
http://http.cs.berkeley.edu/~dfb/papers/thesis.ps

[DGC94] Jeffrey Dean, David Grove, Craig Chambers: *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*.
In Proceedings of the Ninth European Conference on Object-Oriented Programming - ECOOP'95 (Aarhus, Denmark), Springer-Verlag, August 1995.

[EGH94] M. Emami, R. Ghiya, L. J. Hendren: *Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers*.
ACM Conference on Programming Language Design and Implementation, 1994.

[MöWi91] Mössenböck H., Wirth N.: *The Programming Language Oberon-2*.
Structured Programming, 12(4), 1991.

[OST] *The Oberon Slicing Tool*.
http://www.ssw.uni-linz.ac.at/Staff/CS/Slicing.html

[PaR93] H. D. Pande, B. G. Ryder: *Static type determination for C++*.
Technical report, LCSR-TR-197, Rutgers University, February 1993.

[Ste98a] C. Steindl: *Program Slicing (1), Data Structures and Computation of Control Flow Information*.
Technical Report 11, Institut für Praktische Informatik, University Linz, 1998.

[Ste98b] C. Steindl: *Program Slicing (2), Computation of Data Flow Information*.
Technical Report 12, Institut für Praktische Informatik, University Linz, 1998.

[Ste99] C. Steindl: *Program Slicing of Object-Oriented Programs*.
Dissertation, University Linz, 1999.

[Wei81] M. D. Weiser: *Program Slicing*.
In Proceedings of the Fifth International Conference on Software Engineering, May 1981.

[ZhR94] S. Zhang, B. G. Ryder: *Complexity of Single Level Function Pointer Aliasing Analysis*.
Technical report, LCSR-TR-233, Rutgers University, July 1994.