Program Slicing for Large Systems

Christoph Steindl[1]

Abstract. *A* program slice *consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a* slicing criterion*. Program slicing [Wei84] uses control flow and data flow information to visualise dependences and assist the programmer in debugging, program understanding, and program maintenance. We describe a program slicing tool for large systems which are built of modules. Two distinctive features of our Slicer are the support for slicing large systems* (intermodular slicing) *and the usage of* user-feedback *during the computation of data flow information (combining static and dynamic information, yielding* semi-dynamic *slices).*

1. Overview

We implemented a program slicing tool for static forward slicing of object-oriented programs written in the programming language Oberon-2 [MöWi91] (for a technical description see [Ste97]). We did not restrict the language in any kind which means that we had to cope with structured types (records and arrays), global variables of any type, objects on the heap, side-effects of function calls, nested procedures, recursion, dynamic binding due to type-bound procedures (methods) and procedure variables (function pointers), and modules.

Program slicing is useful for debugging [Lyl84], because it potentially allows one to ignore many statements in the process of localising a bug. If a program computes an erroneous value of a variable *x*, only the statements in the *slice* with respect to *x* have (possibly) contributed to the computation of that value; all statements which are not in the slice can safely be ignored.

Weiser [Wei84] originally defined a slice with respect to a program point *p* and a subset of the program variables *V* to consist of all statements in the program that may affect the values of the

---

[1] Institut für Praktische Informatik, Johannes Kepler Universität Linz, Austria
http://www.ssw.uni-linz.ac.at/Staff/CS.html

variables in *V* at point *p*. He presented algorithms which use data flow analysis on control flow graphs to compute intraprocedural and interprocedural slices. Ottenstein and Ottenstein [Ott84] defined a slicing criterion to consist of a program point *p* and a variable *v* that is used or defined at *p*. They computed a slice using a graph reachability algorithm on the program dependence graph. Horwitz et al. [HRB90] developed an interprocedural program representation, the *System Dependence Graph (SDG)*, and an efficient two-pass slicing algorithm that uses the SDG. Their two-pass algorithm computes more precise interprocedural slices than previous algorithms because it uses summary information at call sites to account for the calling context of procedures.

The underlying data structures of our *Slicer* are the *abstract syntax tree (AST)* constructed by the front-end of the Oberon compiler [Cre90]. Additional information (such as control and data dependences) is added to the nodes of this syntax tree during the computation. We define a slice with respect to a node of the AST (starting node). The nodes of the AST represent the program at a fine granularity (see Fig. 1), i.e. one statement can consist of many nodes (function calls, operators, variable usages, variable definitions, etc.). The target and origin of control and data dependences are nodes of the AST, not whole statements. This allows for fine-grained slicing (cf. [Ern94]), therefore we call our slicing method *expression-oriented* in contrast to *statement-oriented* slicing. Our slicing algorithm is based on the two-pass slicing algorithm of Horwitz et al. [HRB90] where slicing is seen as a graph-reachability problem and on the algorithm of Livadas et al. [LivC94, LivJ95] for the computation of transitive dependences of parameters of procedures. In order to slice the program with respect to the starting node, the graph representation of the program is traversed backwards from the starting node along control and data dependence edges. All nodes that could be reached belong to the slice because they potentially affect the starting node.

We extended the notion of interprocedural slicing to intermodular slicing. Information that has been computed once is reused when slicing other modules that import previously sliced modules. Furthermore, we support object-oriented features such as inheritance, type extension, polymorphism, and dynamic binding. Since the construction of summary information at call sites is the most costly computation, it is worthwile to cache this information in a repository and reuse as much information as possible from previous computations.

Zhang and Ryder showed that aliasing analysis in the presence of function pointers is NP-hard in most cases [ZhR94]. This justifies to use safe approximations since exact algorithms would be prohibitive for an interactive slicing tool where the maximum response time must be in the order of seconds. Our approach to reach satisfying results is to use feedback from the user during the computation of data flow information. The user can for example restrict the dynamic type of polymorphic variables and thereby disable specific destinations at polymorphic call sites.

Section 2 discusses intraprocedural and interprocedural control flow analysis that copes with dynamic binding. Section 3 shows how we compute data flow. Section 4 describes our support for interprocedural and intermodular slicing and how we support object-oriented features. Section 5 gives examples of the user interface and user feedback. Finally, section 6 compares our Slicer with other slicing tools and classifies it according to the criteria used in Hoffner´s comparison of program slicing tools [Hof95], whereas section 7 gives an outlook onto future work.

2. Control Flow Analysis

2.1. Intraprocedural Control Flow Analysis

In general, control dependences can be defined in terms of the control flow graph (consisting of so-called basic blocks) and dominators [Aho86] (for a definition of these terms see [FOW87] and [Aho86]). In Oberon-2, the computation of intraprocedural control flow information is - in most cases - very easy, since Oberon-2 contains mainly constructs for structured control flow. These control dependences therefore simply reflect the program's nesting structure.

2.2. Interprocedural Control Flow Analysis and the Handling of Dynamic Binding

For interprocedural slicing, we have to compute interprocedural control dependences due to procedure and function calls which represent transfers of control from the call site to the called procedure. For statically bound calls (ordinary procedure and function calls), the interprocedural control flow graph is built by simply linking the call sites (*call* nodes of the AST) with the destination of the calls (*enter* nodes). In addition to static binding, Oberon-2 offers two ways of dynamic binding: type-bound procedures (or methods) and procedure variables (or function pointers).

For dynamically bound calls we use the following approximation: In the first step we introduce links between the call sites and all possible 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 links (see Section 5). Finally we perform data flow analysis on the selected interprocedural links. Zhang and Ryder showed that aliasing analysis in the presence of function pointers is NP-hard in most cases [ZhR94]. It is therefore reasonable to use approximations during data flow analysis. We decided to use only safe approximations in the first place, and let the user opt for further optimisations (which may be unsafe).

Above all we consider it important to minimise the number of dynamically bound procedures and the number of possible destinations per call site. In order to achieve this we use the following rules:

1. Super calls can be bound statically since it is known at compile time which is the overridden method in the super class.
2. Type-bound procedures (methods) can be bound statically if
   a) the actual receiver of the method is no polymorphic variable,
   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.

For the cases of true dynamic binding we have to find all possible destinations of the calls. For type-bound procedures, these are simply the methods of the statically known class of the receiver and all subclasses. For procedure variables, we have to determine the set of possible destinations either by propagating the assigned procedures along all possible paths in the invocation graph [EGH94] or by using the following approximations:

1. A procedure must be assigned somewhere to a procedure variable. Otherwise it can never be the destination of a call.
2. The parameter lists of the procedure and the procedure variable must match (basically same number of parameters with same types, see [MöWi91]).

In order to verify if these approximations yield reasonable results, we counted the call sites in various software packages and classified them:

Package	$stat_1$	$stat_2$	$stat_3$	$stat_4$	dyn_1	dyn_2	dyn_3
Program Slicer	4290	0	135	22	0	121	51
Compiler	3766	0	0	0	0	0	0
Vector Graphic Editor	899	40	0	0	0	456	14
User Interface Dialogs	1204	67	0	0	0	650	10
Oberon System	5555	0	58	1	0	268	109

Tab. 1. Statistics of statically and dynamically bound calls

$stat_1$: statically bound procedure/function calls
$stat_2$: statically bound super calls
$stat_3$: statically bound method call (receiver = non-polymorphic variable)
$stat_4$: statically bound method call (non-extended record type or non-overridden method)
dyn_1: dynamically bound method call (non-exported record type, all possible methods known)
dyn_2: dynamically bound method call (exported record type, unknown methods may be called)
dyn_3: dynamically bound procedure call via procedure variable

Even for truly object-oriented programs the percentage of dynamically bound calls to statically bound calls does not exceed 30%. This verifies our main assumption that - although dynamic binding cannot be ignored - the majority of calls can be bound statically.

3. Data Flow Analysis

The computation of data flow information is much more complicated than the computation of control flow information. As mentioned earlier, aliasing analysis in the presence of procedure variables is NP-hard for languages like Oberon-2 with arrays, pointers and reference parameters.
We perform the computation of data flow information in two phases: First we compute the used and defined variables per node in the AST. Then we compute the reaching definitions in a syntax-directed manner.

3.1. Computation of Used and Defined Variables

In the first phase we compute the used and defined variables per node in the AST. In Oberon-2 the value of variables can only be changed by assignment statements or by passing the variable as a reference parameter to a procedure, provided that the called procedure actually defines the variable. Then the sets *gen* and *kill* are computed for each node:

gen(node) = {node}   kill(node) = set of all definitions of the variable defined at node - gen(node)

For each variable in used or referenced in a procedure we add additional formal parameter nodes to the procedure’s parameter list and additional actual parameter nodes to all call sites of this procedure. *Parameter-in* and *parameter-out* edges are also inserted between these additional formal and actual parameter nodes with the following restrictions: a *parameter-in* edge is only inserted between the actual and formal parameter node if the formal parameter is at all used by the called procedure. *Parameter-out* edges are only inserted for reference parameters between the formal and actual parameter node if the formal parameter is at all defined by the called procedure. If the formal parameter node is defined on all control flow paths, this definition is a killing definition, otherwise it is a non-killing definition. The conservative assumptions for external procedures (procedures from modules that have not yet been sliced) are: all parameters are used, all reference parameters may be changed and all reference parameters depend transitively on all parameters. Example 2 shows how information about additional parameters is presented to the user.

Example 2:

```
MODULE Sample2;

IMPORT In, Out;

   VAR i: INTEGER;

PROCEDURE Init ((* VAR i: INTEGER *));
BEGIN In.Int(i)
END Init;

PROCEDURE Do ((* VAR i: INTEGER *)); (* dead code, procedure never called *)
BEGIN
   Init(  (* i*));
   Out.String("Hello world!")
END Do;

END Sample2.
```

The comments in *Sample2* are annotations inserted by the *Slicer* and indicate that

- *Init* uses the global variable *i* and therefore has an additional formal parameter node.
- *Do* uses the global variable *i* by calling *Init* and therefore has an additional formal parameter node and an additional actual parameter node when calling *Init*.
- *Do* represents dead code as it is never called and not exported.

3.2. Computation of Reaching Definitions

In the second phase we compute the reaching definitions in a syntax-directed manner. The following data flow equations are used to the compute reaching definitions *in* for each node of the AST [Aho86]:

(1) Sequence S = S1 S2
   gen(S) = gen(S2) ∪ (gen(S1) - kill(S2))
   kill(S) = kill(S2) ∪ (kill(S1) - gen(S2))
   out(S) = gen(S) ∪ (in(S) - kill(S))
(2) Selection (IF, CASE)
   gen(S) = union of gen of all branches
   kill(S) = intersection of kill of all branches
   in of each branch = in(S)
   out(S) = union of out of all branche
(3) Iteration (LOOP, WHILE, REPEAT)
   gen(S) = gen of nested statement sequence S1
   kill(S) = kill(S1)
   in(S1) = in(S) ∪ gen(S1)
   out(S) = out(S1)

The sets *gen* and *kill* have already been computed for each node and all statement sequences during the first phase. In the second phase, *out* is computed by means of the data flow equations given above during a second traversal of the AST.

In Figure 1, we show the AST of the statement sequence in Example 3 with encircled definitions of variables and data dependences to the reaching definitions. The left-hand side of an assignment statement (i.e. the defined variable) is finally data dependent on all used variables of the right-hand side of the assignment. These variable nodes on the right-hand side are again data dependent on all reaching definitions. *if* nodes are data dependent on all variable usage nodes in the expression subtree.

Example 3:

```
Read(i);
IF i < 0 THEN i := -i END ;
j := i * 3;
```

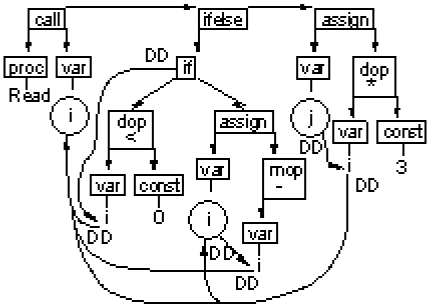


Fig. 1. AST of the statement sequence in Example 3 with definitions of variables, reaching definitions, and data dependences (DD)

4. Expression-Oriented Program Slicing

We perform slicing on a "per node" basis which is also termed *expression-oriented* slicing (in contrast to *statement-oriented* slicing). The target and origin of control and data dependences are nodes of the AST, not whole statements. The units that are considered for inclusion into the slice are AST nodes. This allows for fine-grained slicing (cf. [Ern94]), i.e. the resulting slice is as precise as possible. E.g., only those variable definitions and declarations are coloured blue, whose variables are actually visited during slicing, in the import list only those modules are coloured blue, that are actually needed somewhere in the slice, etc.

4.1. Interprocedural Program Slicing

We modelled our internal data structures for interprocedural slicing after the algorithm of Horwitz et al. [HRB90] and their *System Dependence Graph*. The following nodes and edges are used to facilitate interprocedural program slicing:

- *formal input parameter nodes* are added to the *enter* node of a procedure P for all used or defined variables and *formal output parameter nodes* for all defined variables, corresponding *actual parameter nodes* at the call sites calling P,

- *parameter-in edges* between actual parameter nodes and formal input parameter nodes, *parameter-out edges* between formal output parameter nodes and actual parameter nodes, additionally for functions: *parameter-out edge* between the *procedure exit node* and the call site,
- *call edges* linking the call sites with the destinations of the calls, and
- *transitive dependence edges* of formal output parameters and for the *procedure exit node* on formal input parameters.

We use an adaptation of Livadas' algorithm [LivC94, LivJ95] for the computation of transitive dependences during data flow analysis by slicing the procedure intraprocedurally starting from the formal output parameters and inserting transitive dependences to all parameters that have been marked during slicing. In Figure 2, the formal input parameter node of *in* is marked during slicing for the formal output parameter node of *out*, therefore a transitive dependence edge is inserted from the formal output parameter node of *out* to the formal input parameter node of *in*. This transitive dependence edge is reflected onto all call sites. There is a data depencence edge incident to the formal input parameter node of *in* which means that the value of *in* is used. Therefore, a *parameter-in edge* is inserted from the actual parameter node to the formal parameter node. The formal input parameter node of *out* is not marked during slicing. This means that the definition of *out* at the formal input parameter node does not reach the formal output parameter node. In other words, *out* is defined on all control flow paths in procedure *Abs*. Thus, the definition of the second parameter is a killing definition at call sites of *Abs*. There is no data dependence edge incident to the formal input parameter node of *out*. This means that the value of *out* is never used. It is therefore not necessary to insert a *parameter-in edge* between the second parameter at the call sites of *Abs* and the formal input parameter of *ou*t, but a parameter-out edge is necessary, since *out* is defined in *Abs*.

Interprocedural slices are then computed by solving a graph reachability problem [HRB90]. To obtain precise slices, the computation of a slice must preserve the calling context of called procedures (especially if there are several call sites of the same procedure), and ensure that only paths corresponding to legal call/return sequences are considered.

The first pass of the interprocedural slicing algorithm [HRB90] determines all nodes that can be reached by a backwards traversal of control and data flow edges but without descending into called procedures. Transitive interprocedural dependence edges guarantee that calls can be side-stepped. In the second pass, the algorithm descends into all previously side-stepped calls and determines the remaining nodes in the slice. The slice is the union of the nodes visited during both passes.

Example 4:

```
PROCEDURE Abs (in: INTEGER; VAR out: INTEGER);
BEGIN
    IF in < 0 THEN out := - in
    ELSE out := in
    END
END Abs;
...
Abs(i, j)
```

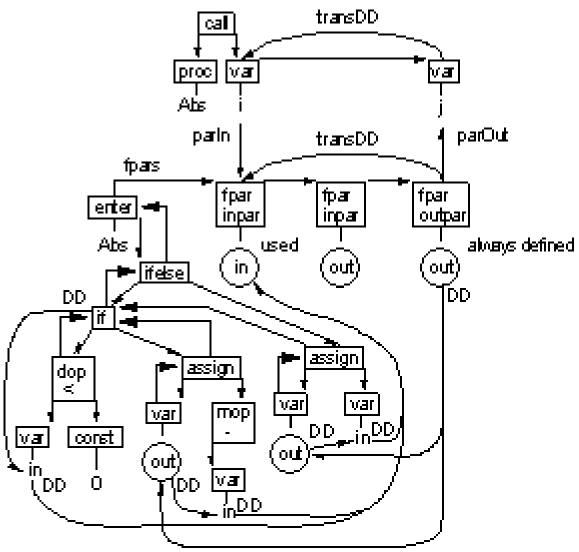


Fig. 2. Computation of transitive dependences (transDD)

4.2. Intermodular Program Slicing

In Oberon, large programs can be split up into several modules which may be compiled separately, each module defining its interface by exporting or not exporting items (variables, constants, types, procedures etc.). This interface is then stored in a so-called "symbol file". Other modules can import already existing modules, and the compiler uses the symbol files for type checking the imported items across module boundaries.

We extended this idea to provide for intermodular slicing in the following way: control flow and data flow information that is computed during slicing is stored in a repository and is reused when slicing dependent modules. The repository is therefore more or less an internalised directory of symbol files. Of course, one cannot keep all the information but has to omit unnecessary details. What is actually made persistent is the following:

- Every exported global procedure (function or method), the module body and every procedure that has been assigned to a procedure variable together with their lists of parameters. Furthermore, we keep the sets of used and defined variables, the additional parameters of every procedure, the transitive dependences of the parameters and information about how the parameters are used by the procedure (parameter used as input parameter, as output parameter, used or defined at all, or defined on all control flow paths).
- Global types (since they may be used by parameters of global procedures, may be exported and extended and so on).

The repository for the core Oberon System (30 modules, 14000 lines, 11000 statements) [WiG88] occupies only about 800 kBytes of disk storage.

4.3. Support for Object-Oriented Features

The symbol table generated by the front-end of the compiler contains all information about classes, their member fields, methods, and the inheritance relationship between the classes. Naturally, the symbol table entry for a derived class only contains information about the newly defined fields and methods. Information about inherited fields and methods can be accessed via a link to the superclass. Dynamic binding is resolved by our interprocedural control flow analysis. *Polymorphic choice nodes* (see section 5.1) provide the user interface to restrict the effects of dynamic binding at polymorphic call sites. We did not need to introduce new kinds of nodes and edges to represent object-oriented features but handled them in a natural way.

5. User Interface and User Feedback

5.1. User Interface

The user interface of our Slicer is currently mainly textual with active elements [MöK96] for the following purposes:

- bi-directional hypertext links between the call sites and all procedures that may be called.
- reaching definitions - all reaching definitions are collected in a popup menu element. By selecting a definition from the popup element, this definition is shown in the code.
- polymorphic choice nodes - the user can enable and disable links to procedures at polymorphic call sites.
- parameter summary information - in the formal parameter lists of procedures parameter summary information is available via parameter info elements. The usage of the parameters is indicated by an arrow (for definition without usage (OUT parameter), for usage without definition (IN parameter), for usage and definition (VAR parameter), to indicate an anomaly for value parameters like "parameter is never used" or "assignment to value parameter", to indicate an anomaly for reference parameters like "possibly not set")). By clicking on these arrows of output parameters, the procedure is sliced intraprocedurally for this parameter.

By clicking on a statement, interprocedural slicing for the corresponding AST nodes is performed.

All source code parts corresponding to AST nodes that have been reached during slicing are then emphasised by colouring them blue. When slicing for the last statement in Example 5, only the parts that are written in bold are actually in the slice. The slice is not executable, but shows all parts that can influence the last statement. The first assignment to *j* is in the slice because it generates a reaching definition for the parameter *j* in *SideEffect(j)*. This parameter is itself a killing definition for *j* because the formal parameter *i* of *SideEffect* is defined on all control flow paths. It is therefore also the only reaching definition of *j* at the last statement. If we were rather interested in an executable slice, everything of Example 5 would be in the slice. Most slicing tools perform *statement-oriented* slicing which would again include everything into the slice.

Example 5:

```
VAR i, j, k, l: INTEGER;

PROCEDURE SideEffect (VAR i: INTEGER): INTEGER;
BEGIN INC(i); RETURN i
END SideEffect;
```

```
BEGIN
   i := F0();
   j := F1();
   k := F2();
   l := i + SideEffect(j) * k;
   j := j
END
```

Furthermore the Slicer inserts annotations into the source code to indicate:

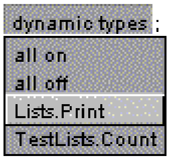
- control dependences and data dependences
- additional parameters for objects in used or defined variables that are not originally part of the formal parameter list
- parameter summary information (e.g.: parameter x is used; parameter y is defined; parameter z is used and always defined)
- dead code (e.g., procedures that are never called or code after a RETURN or EXIT statement)

Each of these active elements and annotations can be turned on and off to avoid getting lost in too much information.

5.2. User Feedback

As mentioned earlier, the user’s expertise is used to restrict the effects of dynamic binding at polymorphic call sites. For type-bound procedures, the user may specify which of the possible dynamic types of the receiver shall be considered during data flow analysis. The notion of "dynamic type" of a polymorphic pointer variable which may point to objects of a given class (corresponding to the static type of the pointer variable) and subclasses thereof has therefore been extended in a natural way to procedure variables (function pointers): For procedure variables, the user may disable some of the possible destinations of the polymorphic call sites.

The user can click on a polymorphic choice element and disable or enable all or some of the destinations of the call. The popped-up view of this element is for example:



The user can disable the link to *TestLists.Count* if she is not interested in the effects of *TestLists.Count* on the polymorphic call site.

6. Comparison

Since the underlying data structure of our *Slicer* is the AST and the symbol table of the Oberon compiler, all object-oriented features like inheritance, type extension, and dynamic binding are already handled and we do not need to introduce additional kinds of edges and nodes like *class entry nodes*, *class member edges* [LaH95a, LaH95b] or *class header nodes*, *class membership edges*, *inheritance edges*, *polymorphic call edges*, and *inherited method edges* [Kri94].

The response times when using our slicing tools are always within 5 seconds for the computation of control and data flow information and less than a second for slicing itself. Memory usage is in the order of some megabytes for large modules, it is approximately twice the memory used by the standard Oberon compiler to compile the module.

According to Hoffner [Hof95], our *Slicer* can be classified as follows:

- *Slicing variables*: The slicing criterion cannot consist of an AST node and an arbitrary variable, rather must the variable be used or defined at the statement.
- *Type of result:* The AST nodes that are part of the slice are emphasised by colouring them.
- *Slicing point*: The slicing criterion is a specific node in the AST rather than a combination of a statement and a variable.
- *Scope of slice*: The scope of the slice is intermodular (not only interprocedural or even intraprocedural). Already computed slicing information is kept in a repository and reused when slicing dependent modules.
- *Slicing direction*: Although our discussion focused on backward slicing, our techniques are also applicable for the calculation of forward slices [HRB90]. No additional analysis information is required to calculate forward slices.

- *Abstraction level*: We use *expression-oriented* slicing which is finer-grained than *statement-oriented* or *procedure-oriented* slicing. The units that are considered for inclusion into the slice are AST nodes.
- *Type of information*: The kind of information is *semi-dynamic* as it combines static information (obtained from the source code) with dynamic information (obtained via user feedback).
- *Computing method*: We compute slices by a graph reachability algorithm on the enriched AST (in contrast to solving data flow equations).

Among other things, Hoffner compares three slicing tools: *Kamkar's slicing tool* [Kam93], *Spyder* [Agr91], and the *Wisconsin Program Integration System*. He reports time measurements between 25 seconds (Kamkar's tool) and 430 seconds (Spyder) for computing control and data flow information of a 300 lines program (Calculator for infix expressions) on a SPARC-station 10/20. The memory usage is estimated by the number of nodes and edges (ranging from about 600 to about 2300 nodes and up to 6000 edges). Our slicer takes approximately one second in order to compute control and data flow information on a 200 MHz Pentium, the time for slicing is not noticeable. During slicing we use approximately 700 kBytes of memory which may be comparable to the memory used by the other slicing tools. Therefore, we dare say that our slicing tool is really appropriate for interactive use.

7. Conclusion and Future Work

Our *Slicer* helps a lot during debugging and reverse engineering. To use Oberon-2 as a source and target language helped us on the one hand (because we already had a parser, a compiler and other tools for that language), but made it difficult on the other hand (e.g., due to reference parameters, dynamic binding, and modules). However, we could not have implemented a slicing tool with the same functionality for as big a language as C++. Therefore we are glad about the choice of the language.

The main contributions of our work are:

- Intermodular slicing with caching of already computed information in a repository.
- Efficient and natural support for object-oriented features.
- Resolution of dynamic binding in the first place and usage of user feedback in order to restrict the effects of dynamic binding later on at polymorphic call sites.

- Expression-oriented slicing yielding precise semi-dynamic slices.

In the future, we will try to further decrease the effects of dynamic binding (cf. [CaG94], [PaR93]). Furthermore, we want to improve our alias analysis to reduce the number of possible aliases at definitions (thereby reducing the number of non-killing definitions).

8. Bibliography

[Agr91] Hiralal Agrawal: Towards Automatic Debugging of Computer Programs. PhD thesis, Purdue University, West Lafayette, Indiana, 1991.

[Aho86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[CaG94] Brad Calder, Dirk Grunwald: Reducing indirect function call overhead in C++ programs. 21st ACM Symposium on Principles of Programming Languages, 1994.

[Cre90] Régis Crelier, OP2: A Portable Oberon Compiler. Technical report 125, ETH Zürich, February 1990.

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

[Ern94] Michael D. Ernst: Practical fine-grained static slicing of optimized code. Technical report MSR-TR-94-14, Microsoft Research.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, Joe D. Warren: The Program Dependence Graph and Its Use in Optimization. ACM TOPLAS vol. 9, no. 3, July 1987.

[Hof95] Tommy Hoffner: Evaluation and comparison of program slicing tools. Technical report LiTH-IDA-R-95-01, Department of Computer and Information Science, Linköping University, Sweden, 1995.

[HRB90] Susan Horwitz, Thomas Reps, David Binkley: Interprocedural Slicing Using Dependence Graphs. ACM TOPLAS vol. 12, no. 1, Jan. 1990.

[Kam93] Mariam Kamkar: Interprocedural Dynamic Slicing with Applications to Debugging and Testing. PhD thesis, Linköping University, Sweden, April 1993.

[Kri94] Anand Krishnaswamy: Program Slicing: An Application of Object-oriented Program Dependency Graphs. Technical report, Department of Computer Science, Clemson University, ftp://ftp.cs.clemson.edu/techreports/94-109.ps.Z

[Lyl84] J. Lyle: Evaluating Variations on Program Slicing for Debugging. PhD thesis, University of Maryland, 1984.

[MöK96] Hanspeter Mössenböck, Kai Koskimies: Active Text for Structuring and Understanding Source Code. SOFTWARE - Practice and Experience, 26(7), 1996.

[MöWi91] Hanspeter Mössenböck, Niklaus Wirth: The Programming Language Oberon-2, Structured Programming, vol. 12, no. 4, 1991.

[Ott84] Karl J. Ottenstein, Linda M. Ottenstein: The program dependence graph in a software development environment. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, April 1984.

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

[Ste97] Christoph Steindl: Program Slicing for Oberon. Technical report 11, Institut für Praktische Informatik, JKU Linz, 1997.

[Wei84] Mark Weiser: Program Slicing. IEEE Trans. Software Engineering, vol. SE-10, no. 4, July 1984.

[WiG88] Niklaus Wirth, Jürg Gutknecht: The Oberon System. Technical report 88, Departement Informatik, ETH Zürich, 1988. An up-to-date version of the Oberon System can be obtained via ftp://ftp.ssw.uni-linz.ac.at/pub/Oberon

[ZhR94] Sean Zhang, Barbara G. Ryder: Complexity of Single Level Function Pointer Aliasing Analysis. ftp://athos.rutgers.edu/pub/technical-reports/lcsr-tr-233.ps.Z