Flow-Sensitive Programming Environments

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

Abstract

Many programmers write their programs with a primitive text editor that has no knowledge about the edited text. On the other hand, they use ingenious compilers that collect control flow and data flow information to perform optimizations and generate optimized code. We show how the programmer can benefit from a programming environment that is flow-sensitive because it has knowledge of the control flow and data flow of the edited programs. Additionally we point out how the programming environment can benefit from the programmer's knowledge.

1 Introduction

Many programmers use primitive editors to write their programs. This approach dates back to the times when we wrote our programs on main-frames and had an ASCII terminal as the front-end. Even back then, compilers invested a great amount of time in the computation of data flow information, e.g. to prevent problems with aliases. The times have changed, we have moved from ASCII terminals to windowed user interfaces. But compilation times have not changed very much and most of the program source code is still just plain ASCII.

There have been some advances, however, to make programming more comfortable. Integrated programming environments nowadays have class hierarchy browsers, a huge amount of online documentation, syntax high-lighting etc. But all this information is either static or only on the syntactical level. Only recently has it been recognized that information from the backend of a compiler could be used as feedback to the programmer in order to make some transformations of optimizing compilers visible and understandable [Bin98].

We have implemented the Oberon Slicing Tool ([OST], [Ste98] and [Ste99]). Program Slicing [Wei84] is a program analysis technique that reduces programs to those statements that are relevant for a particular computation. A slice provides the answer to the question "What program statements potentially affect the value of variable *v* at statement *s*?" Mark Weiser introduced program slicing because he made the observation that programmers have some abstractions about the program in mind during debugging. When debugging a program one follows the dependences from the erroneous statement *s* back to the influencing parts of the program. These statements may influence *s* either because they decide whether *s* is executed or because they define a variable that is used by *s*. Program slicing computes these dependences automatically and thus assists the programmer in a lot of error prone tasks, such as debugging, program integration, software maintenance, testing, and software quality assurance.

We visualize the information gathered by the Oberon Slicing Tool with *Active Text Elements* ([Szy92], [MöKo96]). The editor framework of the Oberon System allows arbitrary objects derived from an abstract base class to be inserted into the text like ordinary characters. These so-called *text elements* are active because they react on mouse clicks. For example, a text element representing a hypertext link will cause the editor to scroll to another text position when the user clicks on the link. Other active text elements may support folding or annotating or they may represent a menu that pops up in reaction to a click.

2 Visualization of Control Flow and Data Flow Information

Information that is obtained via control flow and data flow analysis is much more useful than information that is gained by fuzzy parsing used by many programming environments to implement syntax high-lighting and pretty printing. If symbol information is collected, type information is available for the

declared entities of the program. Control flow analysis can detect dead code. It can further determine the exact call relationships among procedures. Data flow analysis computes which definitions reach which parts of the program. The following subsections describe how we visualize the computed information

2.1 Data Dependences

Data dependences are links from the usage of a variable to all definitions of the same variable that might reach the usage. Data dependences are visualized by small buttons (see Fig. 1) that behave like popup menus showing all reaching definitions. One can select a reaching definition which is then highlighted in the program source code. Potential problems (e.g. usage of a potentially uninitialized variable) are also made obvious to the programmer.

In order to visualize data dependences, exact data flow information has to be computed. This is not possible by simply parsing the source code on a syntactical level. One has to pay attention to the smallest details of the programming language (like short-circuit evaluation of Boolean expressions, side-effects of functions within expressions, scoping, etc.).

2.2 Links Between the Call Site and the Called Procedure

The call sites and the called procedures are linked with bidirectional hypertext links (see Fig. 1). Again, one can select a call destination or call site from popup elements, leading to the respective procedure or call site being highlighted in the program source code. Direct and indirect recursion due to static and dynamic binding are handled. For dynamically bound procedure calls due to method calls or calls via function pointers, all possible call destinations are computed.

2.3 Parameter Usage

For each parameter of a procedure, the information about its usage is computed and visualized properly. At each call site, a small arrow element is inserted before each actual parameter indicating the usage of the parameter within the called procedure: a downward arrow for input parameters whose value is used in the called procedure, an upward arrow for output parameters that are defined by the called procedure, an up and down arrow for parameters whose values are used and that are defined. In the formal parameter list of each procedure, even preciser information is available about each parameter. By clicking on the respective arrows, a description of potential alias problems and data flow problems is displayed.

For dynamically bound procedure calls, the usage of each parameter is combined over all possible call destinations (see Fig. 2).

2.4 Additional Parameters

If a procedure accesses non-local variables that are not part of the parameter list, these variables are added to the (formal and actual) parameter lists as additional parameters (see Fig. 2). This makes the total set of possibly used and changed variables of a procedure call obvious to the programmer at one glance. If the programmer is not aware of the additional parameters he will often overlook these side effects.

2.5 Aliases and Dynamic Types

Two variables are aliases if they may denote the same memory cell. Not only unexperienced programmers are subject to problems with aliases, so it can be a big help to make potential aliases visible as early as possible. In object-oriented programming languages, pointers may refer to objects of different types. It is not always obvious to the programmer what dynamic types a variable may have. Therefore, we collect the set of potential dynamic types into another kind of popup elements. We have extended the notion of dynamic types also to function pointers where the dynamic types represent the set of possibly called procedures (see Fig. 2).

```
MODULE Test;

IMPORT In, Out;

CONST
   TAB = 09X;
TYPE
   Node = POINTER TO NodeDesc;
VAR
   head: Node; global: INTEGER;
TYPE
   NodeDesc = RECORD i: INTEGER; next: Node END ;

   PROCEDURE Push (↓ i: INTEGER    (* VAR ↕ head: Node *)); called at
      VAR n: Node;
   BEGIN                                  data dependences
      NEW(n);
      n.i := i DDs;
      n.next := head DDs;
      head := n DDs
   END Push;

   PROCEDURE Insert* ((* VAR ↕ head: Node; VAR ↓In.Done: BOOLEAN *));
      VAR i: INTEGER;
   BEGIN
      In.Open calling;                                  bidirectional
      REPEAT                                            links between
         In.Int(↕i DDs!) calling;                       call site and
         IF In.Done DDs THEN                            destination
            Push(↓i DDs!  (* ↕head DDs *)) calling
         END
      UNTIL ~In.Done DDs
   END Insert;

   PROCEDURE Show* ((* VAR ↓ head: Node *));
      VAR n: Node; i: INTEGER;
   BEGIN
      n := head DDs;                          two reaching
      i := 1;                                 definitions (one for
      WHILE n DDs # NIL DO                    first iteration and
         Out.String(↓"Position ") calling;    one for all successive
         Out.Int(↓i DDs, ↓0) calling;         iterations)
         Out.String(↓": ") calling;
         Out.Char(↓TAB) calling;
         Out.Int(↓n DDs.i, ↓0) calling;
         Out.Ln calling;
         n := n DDs.next
      END
   END Show;

END Test.
```

Fig. 1 - Data dependences and links for procedure calls

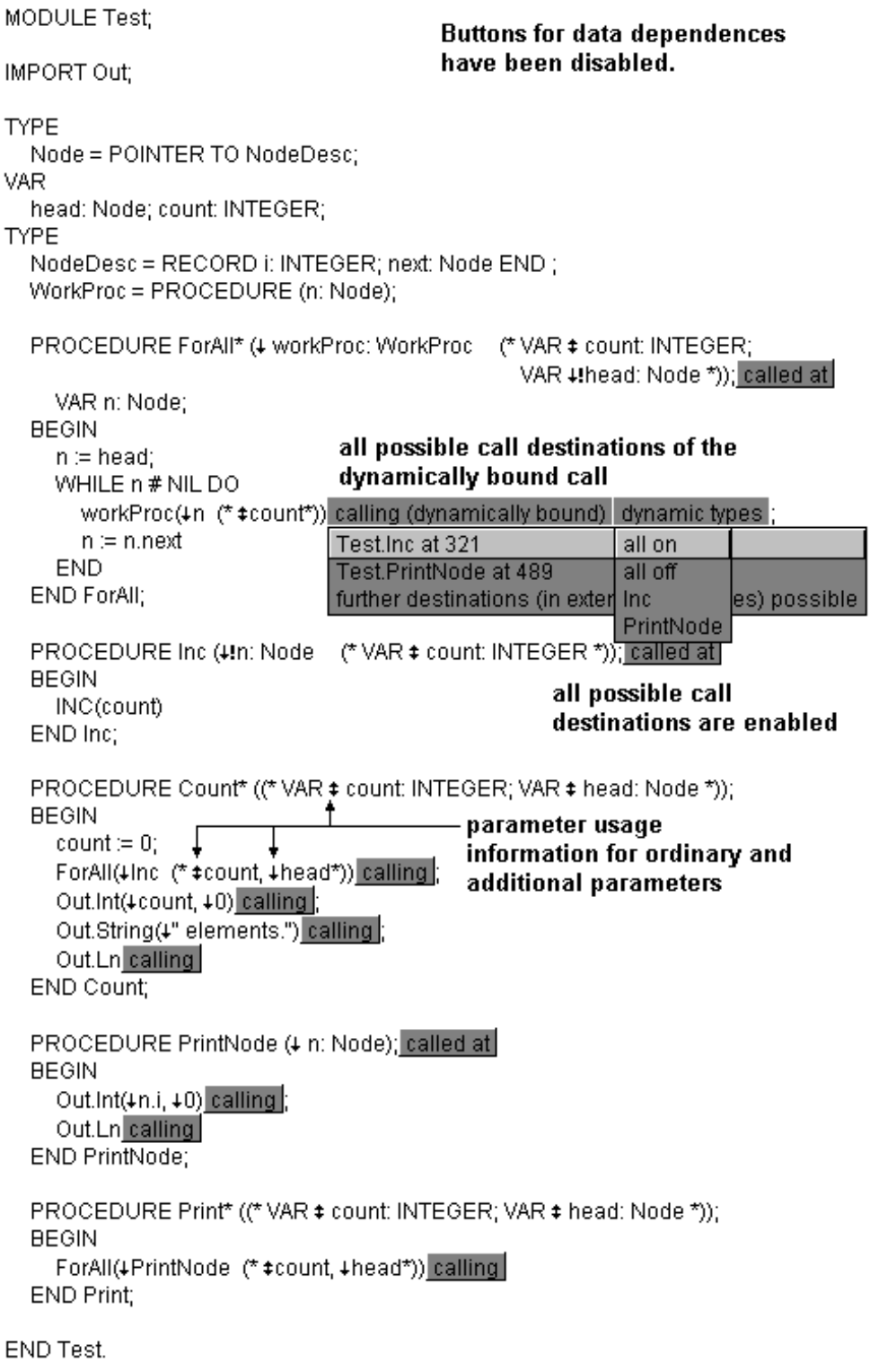

Fig. 2 - Parameter usage, additional parameters, dynamic types

3 Program Slicing

The user can request the computation of a program slice with respect to some point of the program by a simple mouse-click e.g. on an assignment statement. All parts of the program that belong to the slice with respect to the statement and the variables used and defined by that statement are then visualized in a different color.

The derived slices are static, interprocedural, expression-oriented, backward slices. Static slicing means that the information needed for deriving the slices is computed by static analysis of the source code of the program and not by monitoring the execution of the program. Interprocedural slicing means that the slices span over all involved procedures. Expression-oriented slicing means that the smallest entities that are considered for inclusion into the slice are expressions and not whole statements or procedures. Backward slicing means that the slices consists of all parts that may have contributed to the investigated statement and not of all parts that may be influenced themselves by the investigated statement.

4 Bidirectional Feedback

So far, the flow-sensitive programming environment with slicing capabilities provides information only in one direction: from the program slicer to the programmer. The interesting information is computed when the source code is displayed. Therefore it is guaranteed, that the information is up-to-date. Furthermore, it is visualized with active text elements that react to user input. One requirement for this approach is that the computation of the control flow and data flow information is very fast. The Oberon Slicing Tool computes this information within a few seconds.

What is totally new in our approach is, that information is again fed back from the user input to the program slicer. The programmer can e.g., enable or disable some potential aliases if he knows (by some oracle) that two variables will never be aliases. This information is then re-used by the program slicer in order to compute preciser control flow and data flow information. Likewise, the programmer can enable and disable some of the potential dynamic types of polymorphic variables, therefore limiting the set of methods that may be called at polymorphic call sites with these polymorphic variables as receivers of the method calls. In Fig. 2, procedure *ForAll* contains a dynamically bound call with two possible call destinations (*Inc* and *PrintNode*). They are both initially enabled, leading to their additional parameters (namely parameter *count* of procedure *Inc*) being shown at the call site. When the programmer disables the call destination *Inc*, control and data flow information are computed only for the restricted set of possible call destinations. Only the relevant additional parameters (i.e. parameter *head*) are shown.

Bibliography

[Bin98] David Binkley, Brennan Jubb, Bruce Duncan, April Wielgosz: *The FeedBack Compiler*. Proceedings of 6th International Workshop on Program Comprehension, IWPC'98, IEEE Computer Society.

[MöKo96] Hanspeter Mössenböck, Kai Koskimies: *Active Text for Structuring and Understanding Source Code*. SOFTWARE - Practice and Experience, 26(7), July 1996.

[OST] *Oberon Slicing Tool*. http://www.ssw.uni-linz.ac.at/General/Staff/CS/Research/Projects/ProgramSlicing/index.html

[Ste98] Christoph Steindl: *Intermodular Slicing of Object-Oriented Programs*. International Conference on Compiler Construction (CC'98), Lecture Notes in Computer Science 1383, Springer-Verlag, 1998.

[Ste99] Christoph Steindl: *Program Slicing for Object-Oriented Programming Languages*. Dissertation, Johannes Kepler University Linz, 1999.

[Szy92] Clemens Szyperski: *Write-ing Applications: Designing an Extensible Text Editor as an Application Framework*. Proceedings of the 7th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'92), Dortmund, Germany, Prentice Hall, March 1992.

[Wei84] Mark Weiser: *Program Slicing*. IEEE Transactions on Software Engineering, 10(4), July 1984.