Benefits of a Data Flow-Aware Programming Environment

Christoph Steindl
steindl@ssw.uni-linz.ac.at
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 argue that program editors should have the same knowledge about the control flow and data flow of a program. Such editors could help programmers to better understand programs and to be more productive. We propose a data flow-aware programming environment that makes the information that the compiler already computed visible. The bidirectional feedback from the compiler to the programmer and back from the programmer to the compiler enables productive programming and faster debugging.

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 [1] ([Ste98], see [Ste98a] and [Ste98b] for a technical description). Program Slicing [Wei84] is a program analysis technique that tries to reduce the program to the relevant parts. It has been proposed to assist the programmer during debugging, but is also very useful in order to understand programs during software maintenance. The Oberon Slicing Tool has been integrated into the Oberon System. Oberon [2] [WG89] is not only a hybrid object-oriented programming language [MoWi91] but also a programming environment, providing garbage collection and dynamic loading of modules. The linking loader reduces the link phase in the usual compile-link-execute cycle. A fast compiler [Cre90] usually compiles modules within a second which makes the Oberon environment efficient for program development.

The Oberon Slicing Tool uses the same internal data structures as the Oberon compiler. It adds control flow and data flow information to the nodes of the abstract syntax tree and to the nodes of the symbol table. All the information that the Oberon Slicing Tool uses to compute programm slices can be visualized and gives the programmer additional insight into the program. On the other hand, the programmer can give feedback to the program slicer via special user interface elements. This feedback from the programmer to the program slicer is then used to compute more precise control flow and data flow information.

The rest of this paper tries to show the development that led from plain ASCII editors via Active Text editors to a data flow-aware programming environment. Section 2 discusses the step from plain ASCII programming environments to programming environments featuring Active Text. In section 3 we describe the differences between a programming environments with Active Text and a data flow-aware pro-

[1] The Oberon Slicing Tool can be downloaded (including all source and documentation) from http://www.ssw.uni-linz.ac.at/Staff/CS/Slicing.html

[2] An up-to-date version of the Oberon System can be obtained via ftp://ftp.ssw.uni-linz.ac.at/pub/Oberon

gramming environment. Section 4 explains the benefits of bidirectional feedback between the compiler and the programmer. In Section 5 we compare our approach with related work. Finally Section 6 presents our conclusions.

2 From Plain ASCII to Active Text

While it is quite common to use different fonts and figures in books or papers, brochures, in order to explain difficult matters, the same techniques are not used in computer programs.

Most programming environments still present programs in plain ASCII. These environments have some obvious disadvantages: everything that shall help to understand the program source code must be representable in plain ASCII. Otherwise it must be kept in additional documents. Accompanying documentation may contain descriptions of the interfaces of the modules. This documentation might even describe the underlying ideas and design decisions that led to the actual implementation, or it may contain figures and tables. But the problem of this approach is that documentation and source code are split and that they both get old and become inconsistent. An advantage of source code in plain ASCII is that it can be immediately fed into the compiler and that many other tools work perfectly well with plain ASCII (e.g., text search tools like *grep* or version control systems like *SCCS* or *RCS*).

Even if a programming environment provides syntax-highlighting, it often stores the source code in plain ASCII format and parses it before it is displayed. It may not be possible to use different colors or fonts for parts of the programs.

It is a major milestone if the programming environment enables and facilitates the usage of different colors and fonts. Mössenböck and Sikora [MöSi97] describe features of the editor framework of the Oberon System that goes even further. The Oberon System provides an Active Text Framework ([MöKo96], [Szy92]) that allows arbitrary objects such as pictures, tables or buttens to be inserted into the text like ordinary characters. These objects are called *text elements* and are derived from an abstract base class *Elem*. A programmer can easily derive his own text elements from *Elem* and insert them into texts, especially also into source code. 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. Before the program source code is compiled, all foldings are expanded and the plain text (without the active text elements) is fed into the compiler. This is an advantage on the one hand, since the compiler does not have to be aware of the active text elements, but also a disadvantage on the other hand, since the compiler cannot exploit the knowledge "hidden" within the active text elements. The described mechanism is language independent and thus allows arbitrary text (including source code) to be enriched by active text elements.

3 From Active Text to a Data Flow-Aware Programming Environment

A programming environment with active text elements is already a big help for the programmer who has to understand source code during program maintenance. However, documentation tends to grow older much quicker than the described program source code. This effect is diminished if the source code and the documentation are kept in the same document, because then the programmer is always aware of the documentation. But still, the programmer has to provide the additional information. He has to insert hypertext links and annotations where appropriate and he has to use foldings with the idea of stepwise refinement [Wi71] in mind. Otherwise the benefits of active text cannot be exploited. Active text is an offer to the programmer. It can be enforced by project guidelines, but it brings some overhead that the programmer has to perform.

A data flow-aware programming environment with slicing capabilities can bring benefits to the programmer without the disadvantages described above. Therefore it is beneficial to have a program slicer for the respective programming language. This program slicer can then be integrated into the program editor, debugger, and into the compiler. The resulting data flow-aware programming environment can provide the programmer with the benefits of active text elements and with the benefits of language-specific analysis. If the editor and the slicer are implemented as frameworks, futhre analysis techniques can be integrated as extensions easily.

In order to give a flavor of the benefits, we now describe some user interface elements of the Oberon Slicing Tool. These visual elements are generated automatically by the program slicer, based upon the computed control flow and data flow information. Figure 1 shows a module with some of the described visual elements.

3.1 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 sim-

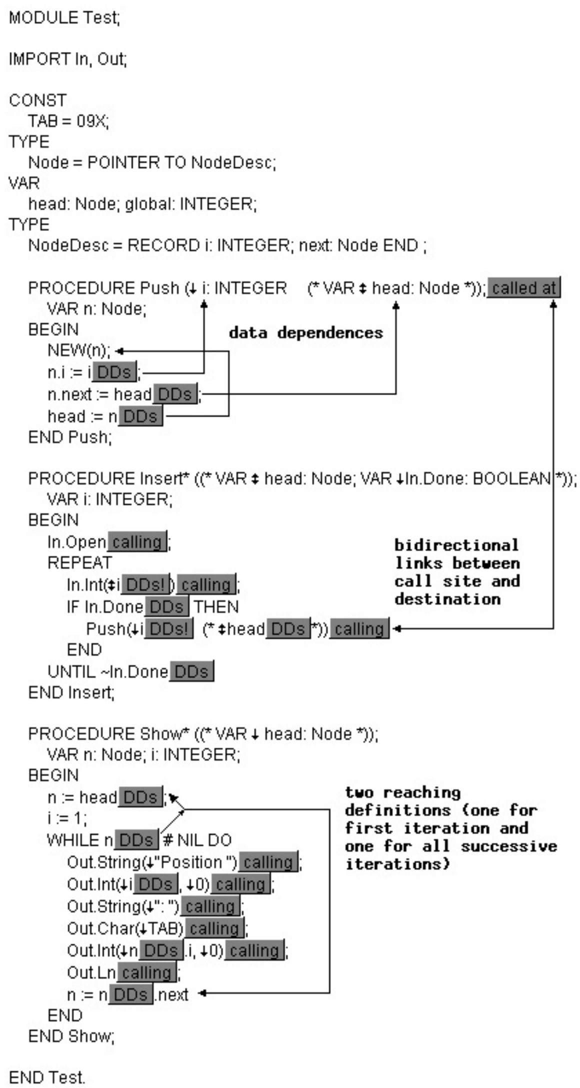

Figure 1: Data Flow-Aware Programming Environment

ply 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 expressions, side-effects of functions within expressions, scoping, etc.).

3.2 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.

3.3 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: downward arrow for input parameters whose value is used in the called procedure, upward arrow for output parameters that are defined by the called procedure, 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).

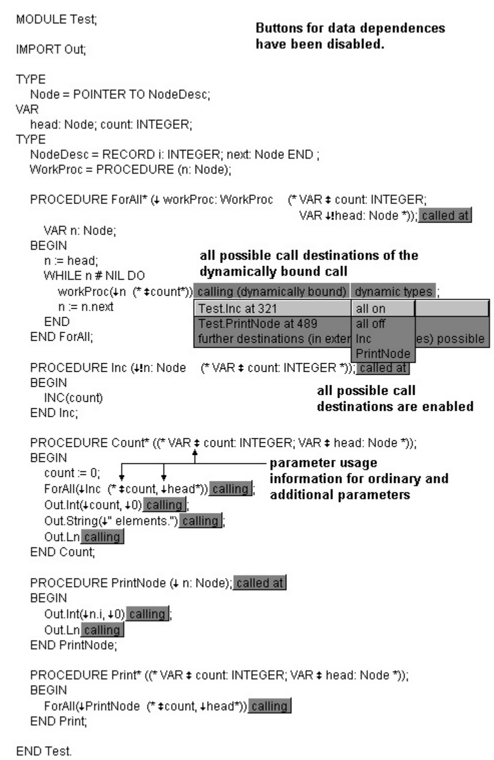

Figure 2: Dynamic Types of Dynamically Bound Calls

3.4 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.

3.5 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).

4 Bidirectional Feedback

So far, the data flow-aware 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 dynamicall bound call with two possible call destinations (*Inc* and *PrintNode*). They are both initially enabled, leading to their additional parameters (namely parmeter *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 are shown (see Fig. 3).

We have noticed that it is quite common during debugging, that the programmer has some use case in mind, that might have led to the error. In order to find the error faster, it

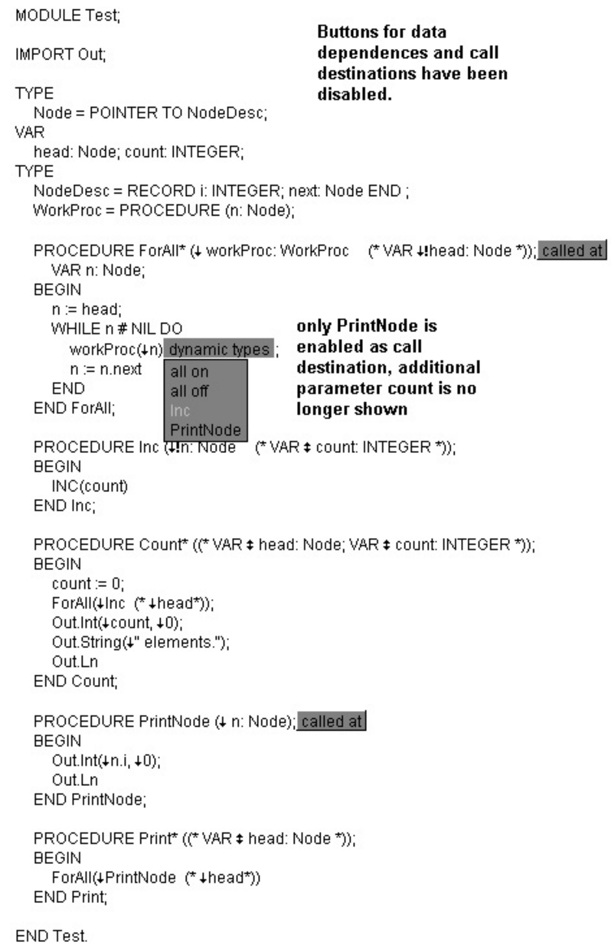

Figure 3: Restricted Dynamic Types due to User Feedback

may be very effective to slice the program for the erroneous statement and then to narrow the program even further by giving feedback about the intended use case to the slicer. The cycle of slicing and feeding back user input to the slicer may continue several times.

5 Comparison

Several systems have been described in the literature that exploit the structure of the source code in order to enable selective reading. These systems can be classified into syntax-oriented editors (e.g., [Zel84] and [HaN86]) and hypertext systems (e.g., [SKP94] and [QuV92]). The former use parsing techniques in order to facilitate syntax-oriented editing, the latter use parsing techniques in order to generate links between syntactically corresponding elements. Another system that is similar to active texts, is described in [Bro86]. A drawback of that approach is that it is not an extensible system.

A data flow-aware programming environment with slicing capabilities and active text elements enhances these existing systems by three major parts: First, it uses advanced techniques to compute control flow and data flow information on the semantic level and not only on the syntactic level. The derived information is visualized with active text elements. Second, user input is fed back to the system and used to derive more precise control flow and data flow information that is no longer restricted by conservative assumptions that are otherwise necessary. Third, program slicing can be used to focus the attention of the programmer to the relevant parts of the program.

6 Conclusions

We have described the benefits of a data flow-aware programming environment with slicing capabilities, where all the information that is used internally to compute control flow and data flow information is also made visible to the programmer. Integration of the program slicer into the programming environment has the benefit that the programmer has access to the advanced information (like aliases and dynamic types of variables) already at compile time. User input is fed back to the program slicer enhancing the accuracy of the slicing information thus reducing the size of the slices. Since the program slicer uses the same internal data structures as the compiler, the feedback from the user can also be used by the back-end of the compiler.

We are using the Oberon Slicing Tool in undergraduate programming courses and are currently waiting for their response. If possible in any way, we want to give qualitative measurements of the impact the data flow-aware programming environment has on their productivity.

We are currently working on an integration of the program slicer with the debugger and the interpreter. Our goal is to provide the slicing information also during debugging and to evaluate whether dynamic slicing can be implemented by extending the interpreter framework [Ob98].

We ses a good chance that next generation programming environments will incorporate advanced techniques to compute exact control flow and data flow information which will then be visualized properly. The step from syntactic information to semantic information will have a positive effect on a programmer's productivity.

Acknowledgements

We would like to thank Prof. Mössenböck and our colleagues at the department for fruitful discussions.

References

[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.

[Bro86] P. J. Brown: *Interactive documentation*, SOFTWARE - Practice and Experience, 16, 1996.

[Cre90] Régis Crelier, *OP2: A Portable Oberon Compiler*, Technical Report No. 125, Computersysteme ETH Zürich, 1990.

[HaN86] A. N. Habermann, D. Notkin: *Gandalf: software development environments*, IEEE Transactions on Software Engineering, 12, 1986.

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

[MöSi97] Hanspeter Mössenböck, Hermann Sikora: *Es muß nicht immer ASCII sein: Aktive Texte in der Praxis und im Unterricht*, Softwaretechnik-Trends, 17:1, 1997.

[MoWi91] Hanspeter Mössenböck, Niklaus Wirth: *The Programming Language Oberon-2*, Structured Programming, vol. 12, no. 4, 1991.

[Ob98] Günter Obiltschnig: *An Object-Oriented Interpreter Framework for the Oberon-2 Programming Language*, Diploma Thesis, Institute for Practical Computer Science, JKU Linz, 1998.

[QuV92] V. Quint, I. Vatton: *Combining hypertext and structured documents in Grif*, Proceedings of the ACM Conference on Hypertext ECHT'92, ACM Press, 1992.

[SKP94] A. Salminen, J. Koskinen, J. Paakki: *HyperSoft: an environment for hypertextual software maintenance*, Proceedings of the Nordic Workshop on Programming Environments Research, Lund, 1994.

[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.

[Ste98a] Christoph Steindl: *Program Slicing (1)*, Technical Report 11, Institut für Praktische Informatik, JKU Linz, 1998.

[Ste98b] Christoph Steindl: *Program Slicing (2)*, Technical Report 12, Institut für Praktische Informatik, JKU Linz, 1998.

[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, Vol. SE-10, No. 4, July 1984.

[Wi71] Niklaus Wirth: *Program Development by Stepwise Refinement*, Communications of the ACM, Vol. 14, No. 4, April 1971.

[WG89] Niklaus Wirth, Jürg Gutknecht: *The Oberon System*, Technical Report No. 88, Computersysteme ETH Zürich, 1988.

[Zel84] M. V. Zelkovitz: *A small contribution to editing with a synax-directed editor*, SIGPLAN Notices, 15 (1984).