DEBUGGING IN APPLICATIVE LANGUAGES

by

John T. O'Donnell and Cordelia V. Hall

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT No. 223

DEBUGGING IN APPLICATIVE LANGUAGES

by

John T. O'Donnell and Cordelia V. Hall

June, 1987

Debugging in Applicative Languages

John T. O'Donnell and Cordelia V. Hall
Computer Science Department
Indiana University
Bloomington, Indiana 47405 USA

Abstract

Applicative programming languages have several properties that appear to make debugging difficult. First, the absence of assignment statements complicates the notion of changing a program while debugging. Second, the absence of imperative input and output makes it harder to obtain information about what the program is doing. Third, the presence of lazy evaluation prevents the user from knowing much about the order in which events occur. Some solutions to these problems involve nonapplicative extensions to the language. Fortunately, the same features of applicative languages that cause problems for traditional debugging also support an idiomatic applicative style of programming, and effective debugging techniques can be implemented using that style. This paper shows how to implement tracing and interactive debugging tools in a purely applicative style. This approach is more flexible, extensible and portable than debugging tools that require modification to the language implementation.

Categories and Subject Descriptors: D.1.1 [**Applicative (Functional) Programming**]: Debugging techniques; D.2.5 [**Testing and Debugging**]: Tracing, Debugging aids; D.2.6 [**Programming Environments**]; D.3.2 [**Language Classifications**]: Applicative languages;

General Terms: debugging, languages

Additional Key Words and Phrases: applicative languages, interactive programs, streams, lazy evaluation.

1. Introduction

Applicative programming languages have a number of apparent advantages over imperative languages. They have clean and simple semantics, are well suited for correctness-preserving program transformation techniques, and their outputs are independent of evaluation order, making them promising candidates for programming massively parallel computers.

These advantages come at a cost: most of the experience we have with imperative languages does not carry over very well to applicative languages. Therefore entirely new programming styles must be developed. This situation is analogous to some of the early efforts at structured programming. Merely telling programmers that they should avoid

goto statements did not show them how to write well-structured programs. Programmers often have a similar reaction to applicative languages: they see a set of missing features (no assignments, no commands, no I/O) instead of a new style for expressing algorithms.

Debugging is a good problem domain in which to study these issues. It is inherently important, since anyone who writes an applicative program will want to get it to work. Debugging also presents several serious problems for the applicative style, and debugging tools are more complex than the typical toy programs that are often used to demonstrate the advantages of applicative languages.

Debugging involves low level activities, such as obtaining information about what a program is doing, and high level activities, such as managing the entire program development process. In this paper, we are concerned with supporting the low level activities, which is prerequisite to implementing higher level programming support. The basic problem is that conventional low-level debugging methods lead to numerous difficulties for applicative languages [4, 8, 9, 14].

There are two opposing approaches for debugging applicative programs. One is to add special features to the underlying language implementation. These extra features often result in nonapplicative extensions to the language. The other approach is to find a way to express debugging algorithms in the applicative style itself. The central thesis of this paper is that *tools for applicative debugging should be implemented in the applicative language itself.*

We will show how to implement several debugging tools in an applicative language. The main result of the paper is an interactive debugging tool that allows the user to observe, control and modify the execution of a program. There is no performance penalty in the parts of the program the user isn't directly interacting with. This tool is implemented purely applicatively.

This paper contains the following sections:

1. Introduction
2. Problems with applicative debugging
3. Alternative debugging methods
4. Overview of Daisy
5. Imperative debugging with lazy evaluation
6. Applicative tracing
7. Order of evaluation and divergence
8. Interactive debugging
9. Conclusion

The first part, Sections 1 through 4, describes the general difficulties with debugging applicative programs, and it outlines a number of methods that have been proposed. Some of these methods involve modifications to the language implementation, and we do not consider them in detail. The second part, Sections 5 through 9, gives a detailed presentation of several debugging approaches that can be implemented directly in the applicative language itself.

Section 2 defines the problem, showing how the restrictions of applicative languages interfere with conventional debugging techniques. Section 3 discusses several solutions that have been proposed, and argues for the advantages of using debugging tools written in the target language instead of modifying the underlying machine. Although our techniques will work in many applicative languages, the examples in this paper are all written in Daisy, which is described in Section 4.

The paper then develops and compares several debugging tools in Daisy. One obvious approach is to graft imperative features onto an applicative language, but Section 5 discusses the resulting problems. Section 6 develops a truly applicative debugging style in Daisy. In this method, the user's program is transformed automatically into another program which produces a trace of execution in an output stream. Although the tracing tool is very useful, Section 7 points out some awkward behavior that results from combining idiomatic programming techniques for lazy languages with a deterministic debugging tool. Section 8 improves on tracing by implementing an interactive debugging tool that does not require a transformation of the entire program. Finally, Section 9 considers the various approaches and discusses their impact on the general usefulness of applicative languages.

2. Problems with applicative debugging

The primary focus in debugging a program must always be on the underlying algorithm, and how the program implements the algorithm. However, debugging also has several mechanical aspects, which must be supported by the programming environment.

1. A program's normal output is sometimes insufficient to isolate bugs, so programmers often insert extra print statements in order to examine intermediate variable values. This technique requires some sort of output facility.
2. Inserting extra print statements into a program and repairing bugs in it both require changes to the program's text. The programming environment must allow the programmer to carry out such modifications.
3. In order to make sense of a program's output, the programmer needs to know something about the order of events that occur during execution. For example, the programmer should know that

   ```
   print x;  print y;
   ```

   will print the value of x before it prints the value of y.
4. A practical programming environment must provide some way for the programmer to break out of infinite (or excessively long) loops and cancel uninteresting output.

Each of these requirements poses a serious challenge for applicative languages, for the following reasons.

1. Imperative output statements cause side effects, and applicative programs don't have any side effects at all. Therefore an applicative program cannot contain any statements

that print intermediate information for debugging (or for any other purpose). The system will simply evaluate the program and print the result.

2. Modifications to the text of a program also constitute side effects, so they are impossible in an applicative programming environment. This problem transcends debugging: if the programmer guesses what is wrong with a program, without using any debugging tools at all, it is still necessary to repair the error without executing any side effects.

3. Lazy evaluation is a crucial implementation technique for most practical applicative languages. Unfortunately, the programmer has little knowledge about when events occur during the execution of a program with lazy evaluation. Even if imperative output statements existed in the applicative language, and even if it were possible to insert them into a program, lazy evaluation would cause the output to appear in a completely unpredictable order. Such output would be worse than useless for debugging. In discussing the development of ML, Milner remarks [8]:

   > "ML does *not* use lazy evaluation; it calls by value. This was decided for no other reason than our inability to see the consequences of lazy evaluation for debugging (remember that we wanted a language which we could use rather than research into), and the interaction with the assignment statement, which we kept in the language for reasons already mentioned."

   Section 5 discusses these issues more fully.

4. The language must provide some sort of nondeterminism in order to allow the user to stop execution at any time. Most languages—both applicative and imperative—lack nondeterministic features.

This paper concentrates on debugging tools which address I/O and lazy evaluation. The broader problems of modifying a program and interrupting infinite loops belong to the programming environment or the operating system, not to the debugging tools. A simple applicative programming environment appears in [10], and a more complex system will be described in [12, 13]. These systems solve the problems of modifying function definitions and interrupting computations, so for the rest of this paper we ignore those issues.

3. Alternative debugging methods

Imperative programming languages have motivated a large amount of work on debugging tools, but comparatively little has been done yet for applicative languages. This is partly because applicative languages are most commonly used for research in language and architecture issues, and they are not yet widely used for applications programming. Most applicative programs that have been published are small, and do not illustrate the difficulties that often arise with large pieces of software. On the other hand, applicative languages will never become popular without practical program development tools.

Several kinds of debugging support have been proposed for applicative languages: