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:

- Use techniques borrowed from imperative languages (see Section 5).
- Claim that applicative programs are so good they don't need to be debugged anyway.
- Return *error values* that tell the user something about how an error occurred [20, 5].
- Trace the execution of an abstract machine model for the programming language.
- Use an exception handler in the language to capture debugging information [6]. Most applicative languages don't have adequate exception handling features for this to work.
- Modify the underlying machine to make it produce debugging information when requested [17].
- Write a meta-circular interpreter for the language in itself, and install debugging facilities in this interpreter [16].
- Develop debugging tools written in the applicative language itself.

We argue that the last of these methods is best, but it is useful first to discuss some of the alternatives.

Most conventional debugging techniques have relied heavily on imperative language features. It is natural to try out such methods in an otherwise applicative language—there is no reason to develop new methods if the old ones are adequate. Section 5 shows why this approach is a dead end.

Many authors have claimed that the good semantic properties of applicative languages make them more reliable and reduce the need for debugging. In our own experience this is true, to a limited extent. That is, we have found that moderately complex applicative programs work correctly the first time more often than equivalent programs in conventional languages (such as Pascal, C, etc.). But that does not mean that debugging is obsolete for applicative languages.

Several applicative languages use *error values* to aid in debugging. The dataflow language Val uses error values to describe what went wrong with arithmetic primitives [20]. For example, there are special values that represent "positive overflow", "negative overflow", etc. In addition, Val provides an algebra that defines combinations of the error values. However, this system is mainly oriented toward handling exceptional numeric conditions, and it doesn't support general purpose debugging.

Daisy also returns error values in many cases. There is a special type for error values, and a primitive function that receives an error value as an argument will generally produce an extended error value. The "unbound identifier" error value `|ubi:var|` indicates that the interpreter tried to evaluate `var` but the environment contained no binding for it. If another function is applied to `var`, the resulting error value will describe the sequence of error propagations. For example, `add : [var 100]` produces `|nn0/ubi:var|`. These error values are useful, but they often become too large to really help the programmer.

A basic problem with error values is that they don't tell the programmer anything unless the system thinks that it has detected an error. If the programmer writes a function that computes a wrong answer, but doesn't generate a numeric overflow, or any other type

of runtime exception, then the function's result will be an incorrect data value rather than a helpful error value. Error values correspond to exception handlers, but they are not general enough for debugging.

An interesting generalization of error values is to make every primitive function return a message that specifies its inputs, even if the inputs are valid and there is no exceptional condition. For example, `add:[x y]` would return `add:[3 4]` if the environment contains the bindings `x=3` and `y=4`. The programming environment could contain two definitions for each primitive: the normal one, and the debugging one. That allows the programmer to switch back and forth between the two sets of primitives while testing the program. (This idea is also useful for providing alternative semantics for a system specification; see [11].) Unfortunately this approach is better at producing massive quantities of output than helping the programmer to pinpoint errors. However, the method of *shadow variables*, described in Section 6, is a useful generalization of this technique.

A good way to debug programs in conventional languages is to trace their execution on an abstract machine. This allows the programmer to watch the program's execution, following assignments to variables, procedure calls and returns, branches, etc. Some of these tools are very sophisticated, and they support concurrent processing and exception handling as well as sequential constructs. Such tools are also useful for teaching beginners how the language works.

Most of the abstract machine models used for applicative languages are poorly suited for direct tracing. The SECD machine is too low-level, and tracing combinator reduction is even worse [18]. Applicative languages derive part of their appeal from clean, machine-independent semantics; debugging tools should not force the programmer to give up those advantages.

A much better alternative is to modify the underlying abstract machine in order to produce debugging output at the level of the source program. Toyn and Runciman have done this for both eager SECD and lazy combinator reduction machines [17]. Their lazy combinator reduction system attaches annotations to expressions as they are reduced. These annotations are ignored during the normal course of the execution. If the programmer interrupts the machine, a debugging system called `glide` uses the annotations to provide a readable snapshot of the current state of the computation. First, `glide` reduces all the reducible combinators (but *not* the reducible primitives). The effect of this is to move data values to their corresponding function argument positions, making the program graph more readable. Second, `glide` traverses the partially reduced graph, using the annotations to describe the function applications and parameter values that are active. After studying the snapshot, the programmer may resume the execution of the program.

The `glide` system allows the programmer to debug at the source language level, without worrying about combinators, and its overhead is relatively low. Since `glide` involves modifications to the abstraction algorithm and the combinator reduction machine, it is not easily portable among language implementations, and the programmer cannot easily make modifications to tailor it for specific debugging situations. However, this

approach appears to be definitely superior to the earlier methods for debugging applicative programs.

Sterling and Shapiro present a meta-circular debugging interpreter for Prolog [16]. This would also work well for applicative languages, and it does not depend on how the implementation works. However, the meta-circular interpreter must still solve all the problems with I/O and lazy evaluation that would appear anyway, and the direct solution we give in Section 8 is much faster.

In Sections 6 and 8 we develop another alternative: instead of developing the debugging tool at the machine level, we write it *in the applicative language itself*—hence the title of this paper. That leads to portable debugging tools which the programmer can easily modify, and it is also interesting to see how debugging tools can be expressed in an applicative style. Furthermore, the `debug` system in Section 8 allows the user to control the execution of a program, and change the values bound to variables while it is executing. Most of the alternatives described above cannot do that.

There are many advantages to a complete programming environment for an applicative language, which is implemented in that same language [1, 19]. The programmer should not have to think about machine-level details of the implementation. Debugging packages should be portable enough to work with interpreters, SECD machines, combinator reduction, native-code compilation, or any other kind of implementation. Finally, it is useful to allow the programmer to modify or enhance the programming environment to meet special needs.

4. Overview of Daisy

Daisy is an applicative language with lazy evaluation based on a suspended list constructor [2]. Daisy implements I/O using streams, and its programming style encourages data recursion and infinite data structures [3]. Since Daisy is similar to other applicative and functional languages, this section just gives a brief overview of its notation. For a complete description of the language, see [5, 7].

Daisy has two types of atomic object: identifiers (like `xyz`) and numerals (like `123`). A list is a sequence of objects in brackets, such as `[a b [1 2] c]`. The programmer can quote an object with the '`^`' character. For example, `abc` will evaluate to the binding of `abc` in the current environment, but `^abc` will always evaluate to `abc`. The elements of a list are written in square brackets, as in `[a [b c] 3]`. Daisy represents a list using the standard cons box method. The fields of a box are called the "head" and "tail". The syntax `[a ! b]` denotes a cons box whose head is `a` and tail is `b`.

Daisy has an explicit `cons` function, but it is usually more convenient to construct a list with "evaluated list syntax", which is a sequence of expressions written in brackets. For example, if `zero` and `one` are bound to `0` and `1` respectively, then `[zero one]` will evaluate to the list `[0 1]`.

Every function in Daisy takes exactly one argument (which may be a list). The infix '`:`' operator denotes function application, so `f:[a b]` evaluates to the result of applying

function `f` to the list of the values of `a` and `b`.

Daisy's conditional expression is similar to that found in many languages: `if` *b* `then` *x* `else` *y* `fi` will produce the value of *x* if *b* evaluates to true and it will produce the value of *y* if *b* evaluates to false. If *b* diverges then the entire conditional expression will diverge also.

An abstraction is written as `\`*formal* . *expression*, where *formal* is an identifier or a list, and *expression* may be an arbitrary Daisy expression. When an abstraction is applied to an argument the interpreter binds the value of the argument to *formal*, and then it evaluates *expression* in the resulting environment. If *formal* is a list, the interpreter automatically probes the value of *expression* in order to find the value of an identifier that it needs. This feature is similar to the use of pattern matching in many functional languages (except that the programmer can specify only one pattern for the *formal*). For example,

```
\[a [b c] d] . add:[b d]  :  [1 [2 3] 4]
```

evaluates to 6.

There are two more ways to bind values to identifiers: `let` and `rec`. The expression

```
let x = inc:2
    y = inc:8
 in add:[x y]
```

will evaluate `add:[x y]` in an environment with `x` and `y` bound respectively to 3 and 9. The interpreter builds a `let` environment nonrecursively; i.e., the right hand sides of the equations are evaluated in the environment that existed before `x` and `y` were bound. The `rec` expression is similar to `let`, except that the right hand sides of the equations are evaluated in the environment that exists *after* the left hand sides are bound. (Lazy evaluation is useful for making this work!) For example,

```
rec f = \n . if zero?:n then 1 else mpy:[n f:dcr:n] fi
 in f:5
```

defines and uses a recursive factorial function.

Data recursion is an important programming technique in Daisy, and the debugging tools use it heavily. The equations in a `rec` expression may be used to define a recursive data structure, just as the previous example defined a recursive function. Thus

```
rec x = [1 2 3 ! x] in x
```

evaluates to a circular list that will print (forever) as `[1 2 3 1 2 3 ...`, but the representation of `x` contains only three cons boxes.

Later sections of this paper contain examples using the following Daisy primitive functions:

`inc :x`	$\equiv$	$x+1$
`dcr :x`	$\equiv$	$x-1$
`add :[x y]`	$\equiv$	$x+y$
`mpy :[x y]`	$\equiv$	$x \cdot y$
`zero? :x`	$\equiv$	$x=0$
`head :[x!y]`	$\equiv$	`x`
`tail :[x!y]`	$\equiv$	`y`

5. Imperative debugging with lazy evaluation

It is interesting to see what happens when the most common debugging technique—inserting imperative print statements into the program—is combined with an applicative language using lazy evaluation. The results would be similar if the system used parallel graph reduction, or any other surprising evaluation order.

The problem is that the program has no understandable evaluation order, so the debugging outputs may be scrambled unrecognizably. This is the main difficulty with programming a lazy system; the programmer cares about the output, not the internal sequence of events during execution.

We illustrate this difficulty with a series of examples. Daisy provides imperative input and output functions, called `console` and `screen` respectively. Of course, pure Daisy programs never use `console` and `screen` except for their interface with the nonapplicative host operating system. However, these functions are quite useful for experimenting with some of the effects of lazy evaluation.

The `screen` function prints its argument on the terminal screen when the application occurs. If a `screen` application is suspended, then the output will not appear until that suspension is probed. This makes it possible to write impure Daisy programs whose output depends on the lazy evaluation order.

Another primitive function called `seq` is often useful for controlling `screen`. (The real purpose of `seq` involves nondeterministic computation, and is beyond the scope of this paper; for example, see [12]). The expression `seq:[a b]` evaluates `a` and then returns `b`. Thus `seq` is equivalent to `if a then b else b fi`.

Using `seq` and `screen`, we can define a function that prints its argument with an identifying tag to help us interpret the output, and then returns its value.

```
print = \[tag x] . seq:[screen:["***" tag x newline] x]
```

Note that `print:x` will produce output only the first time that the program demands its value.

```
let x = print:["x" 5] in [x x x]
     [ *** x 5
     5 5 5]
```

Now consider a function `build` which imperatively prints its arguments and constructs them into a list.

```
build = \[a b] . [print:["alpha" a] print:["beta" b]]
```

In a sequential imperative language with left to right evaluation of arguments, any expression containing `build:[1 2]` would always print `1` and `2` before constructing the list. With lazy evaluation, the imperative `print` applications will not produce any output until the corresponding piece of the list is needed by the rest of the program. If we just ask for the result of a `build` application, the printer will traverse the data structure from left to right, producing the same output that we would expect in an ordinary imperative system.

```
build:[1 2]
    [ *** alpha 1
    1  *** beta 2
    2]
```

Note that the system printed the first token of output (the '`[`') before it probed the head of the list. The `print` outputs "`*** alpha 1 *** beta 2`" appear interleaved with the normal result `[1 2]`.

The following examples show that the imperative outputs only appear when the program needs the corresponding parts of the data structure:

```
head:build:[3 4]
    *** alpha 3
    3
head:tail:build:[5 6]
    *** beta 6
    6
let  x=build:[7 8]  in  add:[head:x head:tail:x]
    *** alpha 7
    *** beta 8
    15
let  x=build:[9 10]  in  add:[head:tail:x head:x]
    *** beta 10
    *** alpha 9
    19
```

These examples have a sequential flavor, but consider what would happen if Daisy `add` expressions (which happen to demand their arguments from left to right) were replaced by very complex expressions, or by nondeterministic computations. This would cause the output to appear in a seemingly random order. Experience has shown that even sophisticated programmers using a lazy language know very little about the actual evaluation order (although sometimes they think they know!).

It is occasionally possible to obtain useful information from imperative output, as long as all outputs are identified by unique tags. However, there are still a number of problems with this approach:

- The debugging output is very hard to understand, since it consists of randomly ordered debugging outputs, interleaved arbitrarily with the correctly ordered "normal" result of the program.
- A parallel implementation would face an even worse problem with interleaving. Suppose that two processors are executing `print:[alpha 1]` and `print:[beta 2]` at about the same time. Unless the system automatically provides mutual exclusion for `screen`, outputs like `alpha beta 1 2` and `alpha beta 2 1` would be possible. This would destroy the usefulness of tagging the output. Since applicative languages are frequently claimed to be appropriate for parallel processors, this is a serious drawback.
- Many applicative languages have nothing like the `screen` primitive. New architectures are being designed specifically for applicative languages, and they may be unable to support imperative hooks just to help in debugging. Therefore the `print` approach is inherently non-portable.
- Imperative output seems like very bad style in an otherwise applicative program. This claim may sound unconvincing, but many applicative programmers find it compelling.

We have found that `print` is sometimes helpful for debugging small programs, but it is certainly not a good foundation for a complete debugging methodology.

There may be a way to integrate I/O primitives like `screen` and `console` into an applicative style. Notice that an interaction between the user and the computer can be described by a program of the form

```
screen : f : console : prompt
```

where the system outputs `prompt` whenever it is looking for input from the user. We could view `screen` and `console` as components of a window on a modern high resolution terminal or work station. The debugging system could open up a new window for interactive debugging at various points in the program selected by the user. Since output caused by different invocations of `screen` would go to different windows, the lazy evaluation order would not cause serious problems. The system could provide a stream of windows, which the program could use in the ordinary applicative style. For example, if a program needs to open a new window, it could evaluate

```
let [[console screen] ! windows'] = windows
 in screen = f : console
```

However, there are numerous difficulties in this approach, and much research is needed for understanding how applicative systems should treat I/O devices.

6. Applicative tracing

An applicative program is an expression that has a value. The system evaluates the expression and prints the resulting value; there is no other way to produce output. Thus the traditional small recursive programs used in teaching Lisp (`factorial`, `append`, etc.) are applicative programs. A debugging tool must provide a sequence of messages for the

user, interleaved with the program's intended output. This section gives several methods for doing that.

The essential idea is to transform a function

$$f : x \mapsto y$$

into a function that returns extra debugging information d in addition to the result:

$$f_{\text{deb}} : x \mapsto [y\ d].$$

This technique appeared earlier in [4]. The debugging value would typically specify the inputs that the function receives, any local computations that it performs, and its final result. We refer to d as a *shadow variable*. By including d in the output of a program we produce a trace, and merely failing to include d in the output expression prevents the lazy system from computing it.

There are two problems to solve, both of which are related to the composition of functions. First, each function must be capable of receiving the debugging values that are embedded in its inputs. Second, there must be some way to output the debugging results of any auxiliary functions. For example, consider the application $g : (f : x)$ where $f : a \mapsto b$ and $g : b \mapsto c$. The transformed functions f_{deb} and g_{deb} produce debugging outputs in addition to their results. But that means that the type of f_{deb} must also be modified to accept the debugging outputs from g_{deb}. Thus we must have $f_{\text{deb}} : [a\,d] \mapsto [b\,d']$ where d is the debugging output produced by the input to f_{deb}, and d' is the debugging output produced by f_{deb} itself. Clearly, d' must contain the information in d as well as a description of what f is doing. Furthermore, any function may be applied to a value produced by some other function. Consequently, all functions must be transformed in this way.

The auxiliary debugging variables d and d' are called *shadow variables*, because the program can examine them to trace its execution, and the program can ignore them if the trace isn't needed. Debugging with shadow variables works very well with lazy evaluation and pattern matching of formal parameters.

Suppose that every Daisy function, including all primitives and all user-defined functions, has the form

$$f : [a_0\ a_1\ a_2\ \ldots a_j] \mapsto [b_0\ b_1\ b_2\ \ldots b_k].$$

Two changes are needed in the debugging version. First, each component of the formal parameter list must be prepared to accept debugging information as well as a value. Second, the output of the function must contain all the debugging information. Therefore the transformed function is

$$f_{\text{deb}} : \left[[a_0\ d_0]\,[a_1\ d_1] \ldots [a_j\ d_j]\right] \mapsto \left[b_0\ b_1\ b_2\ \ldots b_k\ [d_0\ d_1\ \ldots d_j\ d_f]\right].$$

The transformed function f_{deb} returns debugging information which contains all the pieces of debugging output d_i for $0 \leq i \leq j$ produced by its input expressions, and also a description d_f of what f is doing with its inputs.

This scheme works very well with lazy evaluation, because the debugging values will never be computed unless the user asks to see them. By placing debugging outputs after all the normal outputs of a function, the caller doesn't even need to know about the debugging outputs. That means that we don't need separate production and debugging versions of the program; the lazy shadow information will not consume much execution resources unless it is needed.

Since Daisy does not enforce static typing, it is perfectly valid for a function to ignore debugging components of its inputs. Thus the type of the program's inputs may depend on whether we output the debugging information. Consider the following function `f`, which accepts debugging inputs and produces debugging outputs.

```
f = \[[a deba] [b debb]] .
  let x = expression1
      y = expression2
      z = expression3
   in let result = expression4
       in let debout = [deba debb x y z result]
           in [result debout]
```

If we simply want to look at the result that `f` returns, the arguments don't need to contain debugging components. That means that `deba` and `debb` will be undefined, but that causes no harm.

```
let [val deb] = f:[ [1]
                    [2] ]
 in p
```

In order to look at the debugging information produced by `f`, we must define its debugging inputs—e.g., the input `[1]` becomes `[1 "constant"]`.

```
let [val deb] = f:[ [1 "constant"]
                    [2 "constant"] ]
 in ["trace: " deb "final result =" val]
```

Unfortunately, we now need a convention that each argument to a function must be in a list, so we cannot just write `f:[1 2]`. Languages that enforce correct types would not allow us to ignore debugging values so freely.

The following example illustrates the use of shadow variables. The recursive factorial function `fact` would normally be written as

```
fact = \n .
  if zero?:n
     then 1
     else mult:[n fact:dcr:n]
  fi
```

We use a debugging version `mult` of the primitive `mpy` function. Since `mult` is essentially a primitive, it doesn't need debugging inputs, but it produces a result and debugging outputs. When we transform `fact` into a debugging version with shadow variables, the result is

```
fact = \[n debin] .
  if zero?:n
     then let result=1
          in [result
              [debin "fact receives" n "returns" result]]
     else let [fres fdeb]=
               fact:[dcr:n
                     [debin "fact receives" n]]
          in let:[mres mdeb] = mult:[n fres]
              in [mres
                  [fdeb mdeb "fact returns" mres]]
  fi

mult = \[x y] .
  let result = mpy:[x y]
   in [result
       ["mult receives" x y "returns" result]]

f = \n .
  let [result trace] = fact:[n []]
   in [trace newline "final result =" result]
```

Computing `f:4` with this program produces the following output:

```
[[[[[[[[[[] [fact receives 4]] [fact receives 3]] [fact receives
2]] [fact receives 1]] fact receives 0 returns 1] [mult receives
1 1 returns 1] fact returns 1] [mult receives 2 1 returns 2] fact
returns 2] [mult receives 3 2 returns 6] fact returns 6] [mult
receives 4 6 returns 24] fact returns 24]

final result = 24]
```

With a moderate increase in complexity, we can greatly improve the output of this program by properly indenting the output and getting rid of all the irrelevant list structure. A naive approach would be just to `append` together all the debugging output lists to flatten the output. However, that would entail recopying all the output, which is unnecessary overhead. We have found that the philosophy of "recopying the output to format it better" often leads to *several* levels of recopying and a devastating degradation in performance.

Instead, we construct the final output stream directly, using cons to attach the next output token onto a *data continuation* which represents the entire future output of the system. The data continuation dc must therefore be another parameter to fact.

```
fact = \[n depth dc] .
  if zero?:n
     then let result = 1
           in [result
               indent:[depth
                       ["fact receives" n "returns" result ! dc]]]
     else rec [fres fdc] =
                 fact:[dcr:n inc:depth
                       indent:[depth ["fres =" fres ! mdc]]]
              [mres mdc] =
                 mult:[n fres inc:depth
                       indent:[depth ["mres =" mres ! resdc]]]
              [result resdc] =
                 [mres
                  indent:[depth ["fact returns" result ! dc]]]
           in [result
               indent:[depth ["fact receives" n ! fdc]]]
  fi

mult = \[x y depth dc] .
  let result=mpy:[x y]
   in [result
       indent:[depth
               ["mult receives" x y "returns" result ! dc]]]

indent = \[depth dc] .
  rec loop = \i . if zero?:i then dc else ["|   " ! loop:dcr:i] fi
   in [newline ! loop:depth]

f = \n .
  rec [result debug_out] =
         fact:[n 0 [newline]]
   in [debug_out newline "final result =" result]
```

Now f:4 produces much more readable output:

```
[[
fact receives 4
|    fact receives 3
|    |    fact receives 2
```

```
|    |    |    fact receives 1
|    |    |    |    fact receives 0 returns 1
|    |    |    fres = 1
|    |    |    |    mult receives 1 1 returns 1
|    |    |    mres = 1
|    |    |    fact returns 1
|    |    fres = 1
|    |    |    mult receives 2 1 returns 2
|    |    mres = 2
|    |    fact returns 2
|    fres = 2
|    |    mult receives 3 2 returns 6
|    mres = 6
|    fact returns 6
fres = 6
|    mult receives 4 6 returns 24
mres = 24
fact returns 24]

final result = 24]
```

It is usually better for the main function to output the tracing information before the result of the target function. This is because the target expression might diverge. If we output the final result before the trace, and there is no final result, then we will never see any of the trace. The following expression correctly defines the debugging output, but all we get is an opening '`[result =`'.

```
let [result debug_out] = fact:[-1 0 [newline]]
 in ["result =" result newline   debug_out]

[result =
```

The problem, of course, is that our definition of factorial diverges on negative inputs. By outputting the trace before the final result, it is easy to see what is happening.

```
let [result debug_out] = fact:[-1 0 [newline]]
 in [debug_out newline  "result =" result]
 [[
 fact receives -1
 |    fact receives -2
 |    |    fact receives -3
 |    |    |    fact receives -4
 |    |    |    |    fact receives -5
```

```
|    |    |    |    |    fact receives -6
|    |    |    |    |    |    fact receives -7
|    |    |    |    |    |    |    fact receives -8
*** interrupt ***
```

When the programmer wants to see the debugging output only if there actually is a bug, it is better to put the normal function result first. The applicative programming environment [13] can output the normal result first and allow the programmer to interrupt the computation to see the debugging output at any time.

7. Order of evaluation and divergence

It is interesting to consider how these tracing programs deal with order of evaluation. Since Daisy uses lazy evaluation (through the suspended list constructor), the programmer normally knows nothing about the actual time when each expression is evaluated—and yet, the tracing programs don't take this into account, imposing an arbitrary evaluation order on the program. Therefore these tracing programs would be unreliable if they were used in a sequential strict language with undefined order of evaluation (such as Pascal or Scheme). If a particular language implementation used right-to-left evaluation, for example, and the subexpressions produced side effects, then the trace would cause those side effects to occur in the wrong order.

To understand how these tracing algorithms behave in a lazy language with no side effects, we must consider two separate issues: the convergence or divergence of an expression, and the value that the expression has if it converges.

Most applicative languages have the "Church-Rosser property", meaning that all evaluation orders which can be used to reduce an expression e will either converge to a unique value v or else diverge. In other words, picking a bad evaluation order may cause an infinite loop, but it cannot lead to a wrong answer. Let us assume temporarily that the debugging version of a program does not diverge (unless the original program does also).

We don't know anything about the "real" order of evaluation, but the important point is that *since there are no side effects, the order of evaluation does not affect the result or the outputs—therefore a debugging tool can assume any convenient order of evaluation without harm.* And there does not even exist a unique "real" evaluation order for an expression, since components of that expression may be demanded in different ways by different programs.

Programmers who are first learning about lazy evaluation sometimes fall into the trap of thinking too hard about when each evaluation takes place. This is a hopeless task, and it is also a pointless one. Applicative languages need their own idiomatic programming style. Just as an advocate of structured programming should think about program structure instead of avoiding goto statements, the applicative programmer needs to concentrate on the value being defined while ignoring the particular sequence of steps the system will use to compute it.

The tracing functions will produce debugging information for all the values that are defined, regardless of whether those values are used. This is called *full tracing*. Since the debugging version of a program doesn't know which values will actually be demanded by the execution, it may try to print irrelevant values. Full tracing leads to three distinct problems: uninteresting error messages, inefficiency, and divergence when the program should converge. We illustrate each of those situations below.

The debugging package may print *uninteresting error messages* when the program defines values that are sometimes erroneous. Programmers accustomed to strict sequential languages don't often deliberately define erroneous or divergent computations. However, this is a common programming style in Daisy. A typical example is the definition of list substructures which might not exist. Using a sequential style of programming, we might write a function that increments every element of a list by writing

```
inclist = \x .
  if nil?:x
     then []
     else [inc:head:x ! inclist:tail:x]
  fi
```

It may seem cleaner to begin the function with one or more equations that describe the structure of the inputs, and then use any of these defined values which happen to be needed to produce the result. The new `inclist` function begins by using an equation with pattern matching to define `h` and `t` to be the head and tail respectively of `x`. If it later turns out that `x` is nil, then the function ignores `h` and `t`, which are erroneous values anyway.

```
inclist = \x .
  let [h!t]=x
   in if nil?:x
         then []
         else [inc:h ! inclist:t]
      if
```

The tracing version of `inclist` would evaluate and print `h` and `t`, regardless of the value of `x`. This would result in output of the form `h = ...error: head:[]...t = ...error: tail:[]...` whenever `inclist` is called with an empty argument. Such irrelevant error messages are not harmful, but the programmer might not want to see them.

This example is too small to show any advantages for the second `inclist`, but in larger and more complex programs this style often leads to more compact and readable definitions. This programming style is of course somewhat controversial, but programmers do use it and a debugging system must be prepared to deal with it.

A related problem is *inefficiency* that results when the debugging package evaluates and prints irrelevant expressions. Consider a tree search program which either finds the value it is looking for at the root of a tree, or looks for the value in one of the subtrees. A tree is represented as a triple $[v\ l\ r]$, where v is the value at the root, and l and r are the left and right subtrees respectively.

```
search = \[key tree] .
  rec loop = \tree .
        let [v l r]=tree
         in let lv=loop:l
                rv=loop:r
            in if nil?:tree
                    then false
                elseif same?:[key v]
                    then true
                elseif less?:[key v]
                    then lv
                    else rv
                fi
   in loop:tree
```

The tracing package will always carry out a full search of both the left and right subtree, even though many of these searches may be unnecessary. This can greatly increase both the execution time and the quantity of output.

Even worse, a programmer may define a *divergent value* which will never be needed. Here is another way factorial could be defined:

```
fact = \[n] .
  let r = fact:[dcr:n]
   in if zero?:n
         then 1
         else mpy:[n r]
      fi
```

The full tracing version of this function will always result in an infinite loop, since the base case is 0 and `fact:0` defines `r=fact:-1` which diverges. However, `fact` only demands the value of `r` when `n` is positive or negative. If `n` is negative, `fact` diverges anyway, and if it is positive then `r` is well defined.

In each of the three situations described above, the tracing version of a function gets into trouble trying to print a value that is not demanded by the program. An obvious way around this would be to introduce some sort of primitive into the language which determines demand. However, there are several compelling reasons for not doing that. In particular, it has bad implications both for the language semantics and for the architecture that implements it. Another reason we avoid that idea is that it violates our debugging philosophy, which is to implement the necessary tools *in* the applicative style, without resorting to nonportable and nonapplicative extensions to the machine.

So we are left with the problem of dealing with expressions that will not be demanded. Several possible approaches are:

- *Let the debugging function print all values, regardless of whether they are demanded.* If a value that is not demanded produces an error message, the programmer can ignore it. If the evaluation diverges, takes too much time or produces too much output, the programmer can cancel the remaining output and go on to the next step in the debugging package. This requires the facilities of the full applicative programming environment [13].

- Modify the format of tracing versions, so that a guard predicate controls the output of each expression. The debugging function prints the value of an expression only if the associated predicate (which may be a function of the inputs and other local values) is `true`. For example, a useful guard for `p` in the definition of `fact` above would be `positive?:n`. These guard predicates could be supplied either manually by the programmer, or, in some cases, by an automatic analysis of the program.

- Instead of using a transformation of the program which outputs a full trace, use an interactive debugging package. This way, no value would be printed unless the programmer asks for it, and no *a priori* decision needs to be made about which values will turn out to be worth looking at during the course of the computation. We believe that this is the best way to proceed, and the next section shows how it works.

The tracing version of a function is complex enough to require automatic construction using a source-to-source transformation function. If programmers had to write all their code in the style of `fact` above, they would never use this technique. We have actually implemented such an automatic transformation system for a subset of Daisy, which converts a set of functions into tracing versions. This causes considerable overhead, but it is often quite helpful. In most cases, the interactive debugging technique described in the next section turns out to be better than tracing.

8. Interactive debugging

The techniques shown in Section 6 use a purely applicative style, but they have several serious limitations. First, they just produce tracing output. It is usually better for the programmer to be able to interact with a running program, examining only those parts of the program which are suspect, and modifying incorrect values in order to recover from errors. Such interaction requires handling input as well as output in an applicative style, and it needs a mechanism for "changing" the values of variables without using prohibited side effects. Another limitation of the tracing method is that the entire program must be transformed into a debugging version. To be really useful, this transformation should be performed automatically, and it results in a large space overhead.

This section introduces a better applicative debugging tool, called `debug`, which solves those problems. This approach uses several techniques that were first developed for a general applicative programming environment [10, 12, 13].

The main ideas behind the interactive debugging package are:

- The debugging package maintains an input stream as well as an output stream, along with the normal results of the target program's expressions.
- The debugging package traverses the original source program, responding to the user's inquiries. However, the debugging package is not an enhanced interpreter; when asked the value of an expression, it uses the standard interpreting machine.
- The main loop in **debug** uses an accumulator to hold the current value of each parameter and local variable of the target program. When the user types a command that changes the value of a variable, **debug** reenters the main loop with that new value for the corresponding accumulator.
- When the user enters a deeper level of interaction, **debug** creates a continuation that will resume the current level, and passes that continuation to a recursive call to **debug** in order to carry out the nested interaction.

Thus the interactive debugging package is analogous to a structure editor: it traverses a data structure (i.e., the target program), producing only the outputs that are requested.

Before showing how **debug** works, it is useful to see what it does. In order to simplify the implementation of **debug**, we assume that all user-defined functions are written in the form

```
fun = \[a0 a1 ... aj] .
  rec x0 = f0:[...]
      x1 = f1:[...]
       ...
      xk = fk:[...]
   in xi
```

It is relatively straightforward to extend **debug** to handle all the language forms, but we do not show the details here.

Now we shall debug a `factorial` function that relies on user-defined `sum` and `product` functions.

```
factorial = \[n] .
  rec x = factorial:[dcr:n]
      y = product:[n x]
      f = if zero?:n then 1 else y fi
   in f

product = \[a b] .
  rec c = product:[dcr:a b]
      d = sum:[b c]
      p = if zero?:a then 0 else d fi
   in p
```

```
sum = \[a b] .
  rec x = sum:[dcr:a inc:b]
      s = if zero?:a then inc:b else x fi
   in s
```

Now we can test these functions by typing `factorial:[5]`, which unfortunately returns 445. We can directly apply `debug` to `factorial`; it isn't necessary to perform a program transformation to create a debugging version. The debugging system must be supplied with the name of a function and its argument values. It prints a message saying that it has "entered" the function, and it also prints the local variables defined within the function:

```
debug:[factorial [5]]

enter  factorial  at depth  0
   with arguments  [n] = [5]
   and variables  [x y f]
```

The debugging package has now entered an interactive dialogue [10] with the user. The prompt '==>' indicates that `debug` is waiting for a command. It is often useful to type `show`, which asks for a listing of the values of all the parameters and local variables.

```
==> show
  n = 5
  x = 88
  y = 445
  f = 445
```

We already knew that `f` would be 445, and the goal is to figure out what went wrong. Since `x` should be equal to `factorial:[4]`, the recursive call failed. However, before examining what happened during the recursive call to `factorial`, it would be useful to pretend that `x` had received the correct value, and see whether `factorial` would then proceed correctly. So we type an `assign` command, which tells `debug` to "change" the value of a variable (we can also "change" the value of an input parameter to the function). Once `x` has been given the correct value, we tell `debug` to recompute the value of `y` with a `redo` command, and then `eval y` will give us the new value.

```
==> assign x 24
==> redo y
==> eval y
125
```

This provides a real clue—even assuming the recursive call had worked correctly, `factorial` still computed $n \cdot x$ incorrectly. So we should figure out why the multiplication got a bad result; there is no point in examining the recursive call to `factorial` yet. The `enter y` command tells `debug` to enter a new interactive dialogue with the computation

of the expression bound to y. The current dialogue with factorial will be suspended temporarily.

```
==> enter y
enter  product  at depth  1
   with arguments  [a b] = [5 24]
   and variables  [c d p]
==> show
  a = 5
  b = 24
  c = 100
  d = 125
  p = 125
```

Comparing the parameters and local variables with the definition of product, we see that d = sum:[b c] = 24 + 100, and yet debug says that d is 125. So the problem seems to be in sum.

```
==> enter d
enter  sum  at depth  2
   with arguments  [a b] = [24 100]
   and variables  [x s]
==> show
  a = 24
  b = 100
  x = 125
  s = 125
```

The inputs to sum are correct, and the first error is in the value of x—which resulted from a recursive call to sum.

```
==> enter x
enter  sum  at depth  3
   with arguments  [a b] = [23 101]
   and variables  [x s]
```

The recursive call received the correct inputs. This is a hint that the base case is wrong—and the definition says that if a is zero, then sum:[a b] is inc:b instead of just b.

Since sum is called so many times during the evaluation of factorial:[5], it makes sense to get out of debug, fix sum and try again. The done command pops out of all the interactive dialogues, and returns a final result.

```
==> done
  back to sum  at depth 2
  back to product  at depth 1
  back to factorial  at depth 0
```

```
result is 445
```

After fixing `sum`, we can quickly try `sum:[100 24]` to see that it is correct (or at least, less seriously incorrect). This returns `124`, and `factorial:[5]` now produces `120`.

The rest of this section shows the implementation of the `debug` package. More details on the full applicative programming environment appear in [10, 12, 13].

The central function is `main_debug`, which carries out an interactive dialogue with the user in the context of a single function application. Each time the user types an `enter` command, the `main_debug` function will be called. The form of the definition is

```
main_debug = \[function_name depth argvalues dis k] .
   ....output stream....
```

The purpose of `main_debug` is to evaluate an application of the function `function_name` to the list of argument values `argvalues`. The "debugging input stream" of commands typed by the user is `dis`, and `k` is a continuation that specifies the outputs that should be produced after the current interaction has ended. The continuation needs to know the result of the application `function_name:argvalues` (which may be modified by the user) and the stream of input commands that have not been used up by the current invocation of `main_debug`. The `depth` parameter is the number of surrounding interactions that have been suspended. The result of `main_debug` is a stream of outputs that contains all the responses to the user's commands.

The `debug` function provides a convenient user interface for `main_debug`.

```
debug = \[function_name args] .
  [newline ! main_debug :
     [function_name          | name of function being debugged
      0                      | debugging interaction depth
      args                   | top level arguments
      parse:keyboard         | debugging input stream
      \[result dis] .        | top level continuation
        ["result is " result newline]]]
```

The initial debugging input stream is simply a stream of commands from the terminal keyboard, and the top level continuation prints the final result of `function_name:args`.

The `main_debug` function begins by extracting the various components of the function being debugged. Since Daisy uses an interpreter, the source code is already available. Systems based on compilation would need facilities for examining the source code and evaluating expressions. Recall that the function being debugged is in the form

```
fun = \[a0 a1 ... aj] .
  rec x0 = f0:[...]
      x1 = f1:[...]
       ...
      xk = fk:[...]
   in xi
```

The main_debug function begins by defining

```
[arglist local_variables local_exprs result_var] =
   extract:function_name
```

which is equivalent to the following definitions:

```
          arglist = [a0 a1 ... aj]
  local_variables = [x0 x1 ... xk]
     local_exprs = [f0:[...]  f1:[...]  ...  fk:[...]]
      result_var = xi
```

Next it defines variables to be the set of parameters and local variables, and defines exprs to be the expressions that correspond to the variables. The initialization of main_debug concludes by computing normal_values, which is the set of values that would be bound to the variables in a normal evaluation of function_name:argvalues. Using this extensive set of definitions, main_debug defines an accumulator-style loop which carries out the interaction with the user. The loop receives a debugging input stream and the current set of values bound to the variables. It defines cmd to be the next command in the debugging input stream, and executes the corresponding expression.

```
main_debug = \[function_name depth argvalues dis k] .
  let [arglist local_variables local_exprs result_var] =
        extract:function_name
   in let variables = append:[arglist local_variables]
          exprs = append:[mk_exprs:arglist local_exprs]
      in let normal_values =
             evalrec:[append:[arglist variables]
                      append:[quote_all:argvalues exprs]
                      variables]
         in rec loop = \[dis values] .
                let [cmd!dis'] = dis
                 in if same?:[cmd newline]
                    then loop:[dis' values]
                 elseif same?:[cmd "...."]
                    then ....
                 elseif same?:[cmd "...."]
                    then ....
                 else ....
                 fi
            in loop:[dis normal_values]
```

The rest of main_debug consists of a set of command implementations of the form

```
elseif same?:[cmd "...."]
   then ....
```

We will only show a few of these in order to illustrate the important techniques.

It is frequently necessary to evaluate an expression in an environment where the **variables** are recursively bound to a corresponding set of **expressions**. The debugging package contains a function called **evalrec** which does this. For example, if

```
 variables  =  [a      b    c]
    values  =  [inc:b  100  add:[a b]]
result_var  =  c
```

then `evalrec : [variables values result_var]` evaluates to 201, since `c = a+b = b+1+b = 100+1+100`. The definition of **evalrec** using Daisy's primitive functions is beyond the scope of this paper. However, any applicative language must have some equivalent evaluation primitive in order to be able to support a programming environment.

The simplest command is **eval**, which tells **main_debug** to read another form from the debugging input stream and evaluate it using the current values of the **variables**.

```
elseif same?:[cmd "eval"]
   then let [exp!dis'']=dis' in
          [evalrec:[variables values exp] newline prompt !
            loop:[dis'' values]]
```

The **assign** command takes two parameters from the debugging input stream: a variable to be modified, and an expression giving its new value. Of course, there is no imperative side effect to the variable. Instead, the implementation of **assign** merely calls **loop** with the new value of the variable substituted in place of the old. The **update** function does *not* execute a side effect to the environment; it merely creates a new environment which is like the old except that **lhs** has been rebound to **value**. The applicative language must implement tail recursion properly for this code to work.

```
elseif same?:[cmd "assign"]
   then let [lhs!dis'']=dis' in
         let [rhs!dis''']=dis'' in
          let value=evalrec:[variables values rhs] in
            [prompt !
              loop:[dis''' update:[variables values lhs value]]]
```

The **redo** command simply evaluates the expression corresponding to the specified variable and updates the value. It is similar to **assign**, except that the user doesn't provide the new value to be assigned to the variable.

```
elseif same?:[cmd "redo"]
   then let [lhs!dis'']=dis in
         [prompt !
          loop:[dis''
```

```
                update:[variables values lhs
                        evalrec:[append:[arglist variables]
                                 append:[argvalues values]
                                 getcode:[lhs variables codes]]]]]
```

Several commands allow the user to enter and exit dialogues, so they must create and use the data continuations. The **ok** and **done** commands return from a dialogue, so the use the **k** continuation. The **enter** command must create a new continuation.

When the user types **ok**, **main_debug** produces a prompt "==>" and then applies the continuation to the current value of **result_var** and the unused inputs. That ends the current dialogue and pops back to the most recently suspended one. Notice that the function does not execute an imperative command to print the prompt; it merely uses **cons** to put the prompt in the output stream.

```
elseif same?:[cmd "ok"]
   then [prompt !
         k:[evalrec:[variables values result_var] dis']
```

The **done** command is similar, except that **main_debug** gives **dis** (whose head is **done**) to the continuation instead of **dis'**, ensuring that the surrounding invocation of **main_debug** will also execute the **done** command. That forces the program to pop out of all the suspended dialogues, back to the top level.

```
elseif same?:[cmd "done"]
   then k:[evalrec:[variables values result_var]
           dis]
```

The **enter** command recursively calls **main_debug** to evaluate the expression bound to one of the local variables. The current invocation of **main_debug** creates a continuation which will resume itself when the user exits from the the recursively entered invocation. That continuation prints a message telling the user that the current level is back in control, and updates the variable that was "entered" with the result of the recursive call.

```
elseif same?:[cmd "enter"]
   then let [lhs!dis'']=dis' in
        let exp = get_expr:[lhs variables exprs]
         let fcn = get_fcn:exp
             argcode = getargcode:exp in
          let args=evalrec:[variables values argcode] in
           main_debug :
             [fcn
              inc:depth
              args
              dis''
              \[result dis] .
                 ["back to" function_name "at depth" depth
```

```
newline prompt !
loop:[dis
        update:[variables values lhs result]]]
```

The actual implementation of the **debug** package is more general and supports a number of additional commands. However, the implementation techniques are similar to the ones above.

The **debug** tool is efficient because

- it requires no source-to-source program transformation;
- it uses the system's own interpreter to evaluate the expressions at full speed, as long as the user does not "enter" them;
- it responds to the user's commands efficiently (for example, extensive output filtering is unnecessary); and
- it produces only the output requested by the user (unlike tracing).

When a programmer wants to explore a program to isolate a bug, **debug** is very effective. The tracing techniques from Section 6 are still useful when the programmer wants to watch the entire execution of a program.

Many enhancements to these debugging tools are feasible. There are several ways to combine tracing with interactive debugging and editing, and there are many more useful commands for the interactive **debug** package. It would also be interesting to experiment with higher-level debugging tools, like Shapiro's algorithmic program debugging methodology [15]. The important point is that all of these extensions are possible because they just require modifications or additions to an existing applicative program. If the debugging tools were built into the language interpreter or a special-purpose applicative language architecture, that would not be the case.

9. Conclusion

Conventional imperative debugging techniques, based on inserting print statements into a program, are not generally useful for applicative languages. It is possible to add debugging features to the language's implementation, but such tools are not portable and the programmer cannot tailor them for special situations. The ideal way to support debugging in an applicative language is to develop a set of tools in that language itself.

The two basic methods for writing a debugging package in a purely applicative style are:

1. Transform the target program into a version where each function returns a pair of the form [*result debug-info*]. The transformed program produces a result and a trace of its execution. The entire program must be transformed.
2. Use an interactive debugging function that traverses the evaluation of the target program, reading commands from the user, executing them, and printing the results.

Streams are used for all input and output. The debugging function uses the system's evaluator to compute the values of expressions unless the user asks to enter an evaluation interactively.

We have found both of these methods to be useful, but the second is preferable in most cases because the user can reduce the amount of irrelevant output by directing the debugging package to the parts of the program that are causing trouble.

There are several advantages in implementing the debugging tools in Daisy (or any other applicative language) rather than in the underlying machine. A user who wants to understand the debugging system deeply can look at the code, and even modify it if necessary. Similar debugging tools can be implemented for any other applicative language. Finally, a library of similar programs is useful for teaching novices how to write applicative programs.

Comparing the ease of debugging in applicative and imperative languages is a subtle problem. It clearly takes more work for a programmer to set up debugging streams in an applicative program than to insert a print statement into an imperative program. This seems to be an inherent disadvantage of applicative programming, but there are two mitigating factors: (1) automatic debugging tools make the process transparent to the user, and (2) the applicative debugging tools work correctly in a parallel environment.

Applicative debugging tools can be automated, either as program transformations (Section 6) or through on-the-fly traversal and interpretation (Section 8). These tools become part of the programming environment, and they are transparent to the user.

The applicative program (complete with debugging streams) is guaranteed to produce exactly the same output on a sequential machine or on a parallel machine. In contrast, consider what would happen to a programmer who inserts some output statements into an ordinary imperative language.

```
function P                    function Q
   x := 3                        y := 4
   print (" x is ")              print (" y is ")
   print (x)                     print (y)
```

Now, this program will always produce the same results on a sequential machine, given the same inputs. However, a parallel implementation might execute applications of P and Q simultaneously, interleaving the output from the print statements and producing any of the following results:

```
x is 3 y is 4
y is 4 x is 3
x is  y is 3 4
x is  y is 4 3
y is  x is 3 4
y is  x is 4 3
x is .... y is 3 .... 4
```

The first of these results is what the programmer expects. The last one is especially damaging, because the programmer may fail to realize that interleaving has led to misleading output.

Of course, mutual exclusion algorithms for preventing such undesirable interleavings of output are well known. However, the imperative programmer using a parallel machine must know when to use mutual exclusion. If it isn't used where necessary, incorrect output will result. If it is used where unnecessary, performance is likely to suffer through "hot spot contention". Either way, the programmer cannot simply insert some print statements and hope for the best.

The applicative program with debugging streams already contains all the synchronization needed for safe parallel execution. The programmer doesn't need to analyze each part of the program to see where mutual exclusion is required; the program explicitly states the exact ordering of all output *even if the program runs on a parallel machine.*

A fundamental decision is whether to support debugging at the machine level or the language level. Turner points out that it is hard to understand a partially evaluated program after it has been compiled to combinators [18]:

> "... run-time error reports are very opaque. Descriptions of the run-time state in terms of the configuration of combinators on the stack are quite unintelligible to users."

The problem here is precisely that the compiled form of the program is at a lower level than the original program. Similarly, a programmer using a high level language should not need to look at machine language, or core dumps, or interpreting machine registers while debugging. The programmer should be able to debug at the same level as the program itself—and this implies that it is better to support debugging in the language, instead of augmenting the underlying machine with special features.

There are many ways to implement a given applicative language: interpretation, compilation to combinators, compilation to host native code, sequential graph reduction, parallel graph reduction, etc. Debugging techniques that use the language's own facilities are portable, and will work regardless of how the implementation works. The advantages of applicative languages stem from their semantic properties, and it is better to exploit those properties while debugging rather than subverting them in order to examine the machine state.

These observations also apply to the entire software environment. An ideal system would consist of a massively parallel architecture supporting an applicative language, with the operating system and programming environment written entirely in that language. Experience with applicative debugging makes this goal seem more plausible.

Acknowledgements

We would like to thank Steve Johnson, whose efforts in designing and maintaining the Daisy system made this work possible.

References

1. Delisle, N. M., Menicosy, D. E., and Schwartz, M. D. "Viewing a programming environment as a single tool", *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments* (Apr. 1984), 49–56.
2. Friedman, Daniel P. and Wise, David S. "CONS should not evaluate its Arguments", *Automata, Languages and Programming* (Michaelson, S. and Milner, R. (ed.). Edinburgh University Press, Edinburgh (1976), 257–284.
3. Friedman, Daniel P. and Wise, David S. "Unbounded computational structures", *Software Practice and Experience* 8 (1976) 407–416.
4. Hall, Cordelia V. and O'Donnell, John T. "Debugging in a side effect free programming environment", *Proc. 1985 SIGPLAN Symposium on Programming Languages and Programming Environments* (June 1985).
5. Johnson, Steven D. "Daisy language manual" (working title), Computer Science Department, Indiana University, Bloomington (1987) (in progress).
6. Kieburtz, Richard B. "A proposal for interactive debugging of ML programs", *Proc. of the Workshop on Implementation of Functional Languages*, Report 17, Programming Methodology Group, Chalmers University of Technology, Goteborg, Sweden (1985) 151–155.
7. Kohlstaedt, Anne T. "Daisy 1.0 reference manual", Technical Report 116, Computer Science Department, Indiana University (1981).
8. Milner, Robin. "How ML evolved", *Polymorphism, the ML/LCF/Hope Newsletter* (Cardelli, L. and MacQueen, D. (ed.) 1, 1 (Jan. 1983) 1–6.
9. Morris, James H., Schmidt, Eric and Wadler, Philip. "Experience with an applicative string processing language", *Conference Record of the Seventh Annual Symposium on Principles of Programming Languages* (Jan. 1980) 32–46.
10. O'Donnell, John T. "Dialogues: a basis for constructing programming environments", *Proc. of the ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments* (June 1985).
11. O'Donnell, John T. "Hardware description with recursion equations", *Proc. of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, North-Holland (Apr. 1987).
12. O'Donnell, John T. "Communication structures for interactive applicative programs", Computer Science Dept., Indiana University (1987) in progress.
13. O'Donnell, John T. "An applicative programming environment",' Computer Science Dept., Indiana University (in progress).
14. Peyton-Jones, Simon L. "Directions in functional programming research", *Distributed Computing Systems Programme* (Duce, D. A. (ed.)), Peter Peregrinus Ltd., London, (1984) 221–245.

15. Shapiro, Ehud Y. *Algorithmic program debugging*, The MIT Press, Cambridge (1983).

16. Sterling, Leon and Shapiro, Ehud. *The Art of Prolog, Advanced Programming Techniques*, The MIT Press, Cambridge (1986).

17. Toyn, Ian and Runciman, Colin. "Adapting combinator and SECD machines to display snapshots of functional computations", *New Generation Computing* 4 (1986) 339–363.

18. Turner, David A. "A new implementation technique for applicative languages", *Software—Practice and Experience* 9 (1979) 31–49.

19. Wertz, H. "An integrated Lisp programming environment", *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging* (1983) 91–95.

20. Wetherell, C. S. "Error data values in the data-flow Language VAL", *ACM Trans. on Programming Languages and Systems* 4, 2 (Apr. 1982) 226–238.