Walk Backwards to Happiness—Debugging by Time Travel

Simon P Booth
Simon B Jones

s.p.booth@stir.ac.uk, sbj@cs.stir.ac.uk

April 1997

Abstract

In circumstances when a variable in a program has an incorrect value the process of debugging it is often a process about discovering the history of that variable, or rather the ancestry of the value that it contains.

In this paper we propose a new technique for debugging that revolves around being able to trace back through the history of a particular variable or variables that it depends upon.

Our development is in the domain of functional programming as the proposal has particular significance in this domain due to the fact that so many standard debugging techniques cannot be used at all.

1 Introduction

Debugging is sometimes an exercise in finding out what happened in the past. We have run our program and it has produced an incorrect result not by failing to execute but by, say, printing the wrong value for a calculation. We now know that the value represented in our source code by, say, `x` is incorrect. The problem now is to find out what has caused `x` to be incorrect. Traditionally the techniques we would now deploy would be a mixture of setting breakpoints or watches or perhaps more simply just printing out what was happening inside the program at various points of, what we hope are, interest. What we might discover is that at some point a variable `a` is involved in some calculation with `x` and that `a` is incorrect. Now the focus of the debugging exercise changes and our attention switches to `a`. So we stop the current execution and then deploy probably the same mixture of techniques described above but with the focus on `a`. Eventually we, hopefully, will successfully trace the error back to its source and fix our program.

It is interesting that the techniques described above have remained largely static for a number of years whilst programming techniques have changed significantly. A programmer from the 1970's would need to learn about object-oriented programming and would no doubt appreciate the Integrated Development Environments (IDEs) now available but would instantly recognise the tools and tricks described above to debug a program! Partly, this is because these techniques do work—we can fix errors in a program's code with these techniques. Equally it must be worthwhile investigating whether better techniques can be discovered.

What we will present in this paper is a proposal for a new 'technique' that we believe captures the behaviour described in the first paragraph but in a much more

sophisticated manner. Our development work so far has been in the functional programming domain; in this domain it is far from clear what the most appropriate debugging techniques are: in a *lazy evaluation* context the control flow may be obscure. In general the conceptual view of a program is based around data dependencies rather than control flow, and we are interested in being able to investigate a program using this conceptualisation, so breakpoints are of little use; 'variables' do not change their values, so watches are meaningless.

1.1 'Backtracing' in the functional domain

The key idea that we present in the paper can best be summarised as 'debugging by dataflow' or 'backtracing'. By this we mean that the programmer when debugging their program can follow the 'story' of how the value under consideration was constructed. To illustrate, consider the following function:

```
f a b = x * y
        where x = a + b
              y = a - b
```

The information we wish to provide the programmer with is how the values are constructed. For instance, if the programmer requests the value of `x`, the detail returned is that `x` is currently 5 and that the value five is the result of adding `a` and `b`. At this point the programmer can request the value of `a` or `b` and see similar detail. Thus the programmer having identified the wrong value can trace its dataflow backwards and trace back to the point where the error was made.

This is what we all do when we debug imperative programs but we do it by setting backpoints and traces to investigate the behaviour of the variable in question. The drawback is that 'traditional' debugging does not capture the 'story' of each value. Thus we cannot go 'backwards' from the point of error. All we can do is restart the program and, via our intuition, attempt to work a little further back towards the actual error. The proposal outlined above would appear to offer considerable benefits to the programmer: finding the source of an error should be considerably quicker and simpler.

1.2 In the imperative domain

To apply the proposal to imperative programs would appear straightforward to describe but rather harder to implement. For the present, we will simply describe how we see this proposal working with an imperative language. To illustrate, we will use the same example from the previous section but rewritten imperatively:

```
x := a + b;
y := a - b;
z := x * y;
```

We'll assume that `z` produces the wrong value. The programmer would mark `z` in same way (in the source code), say, by adding an `*` or some other 'special' character. When the compiler is then run (in debugging mode) upon encountering the marked variable(s) would perform a dataflow analysis to see which variables the marked variable(s) was dependent upon. All the assignments that involved variables that the marked variable(s) were dependent upon would then be modified to return not only the value in the normal way but just as described above the 'story' would

be returned too. In the example, `x` would have the value (5, a{3} + b{2}). Again it would be possible to then inspect `a` to find its 'value'.

The structure of this paper is the following: In section 2 we present the mechanics of the proposal—our initial development will be in the functional domain (as the proposal seems to provide a very useful debugging mechanism in a programming domain where debugging is notoriously difficult). Section 5 is a brief survey of related work (mainly from the functional domain—where the proposal has been examined by other researchers but without the particular emphasis that we place on how the information collected can be exploited to work backwards). Lastly, section 6 will be conclusions and suggestions for further work.

2 Recording backtrace dependences at run time

We can illustrate the basic information to be gathered during program debugging using the following definition of the factorial function (in Gofer):

```
fact    ::  Int -> Int
fact 0  =   1
fact n  =   n * fact (n - 1)
```

Our starting point for tracing back through the construction of bad values, to determine the origin of the error, is a complete trace of the execution of a program (up to the run time failure point if that is what happens). We require that the trace allows us to recover easily not only what data *values* contributed to the construction of any faulty value, but also, recursively, the detailed origins of those data values. Therefore, the general principle of the scheme is that each *value* in a program execution is 'tagged' with a 'story' describing its origin, and each story will, recursively, contain the stories of the values that it depends on. Whenever a value is used in a computation (basic operator or program defined function application), its story is built into the story tagged onto the new value constructed. Further, where a value is passed as an argument to a function application, it is passed *complete with its story* so that any values computed in the function's body can be tagged with the full story of their origin; other approaches do not make this generalization, and so lose the ability to explicitly trace back from any faulty value to its origins.

For example, if we apply `fact`, above, to `1`, we would like to be able to derive the following explanatory facts:

1. The result is `1`, from application of `fact` to the value `1` (which was initial data).
2. The result is `1`, obtained by multiplying `1` (which was initial data) by `1` (which was obtained from application of `fact` to `0`).
3. The `0` in 2. was obtained by subtracting `1` (a constant in `fact`) from `1` (initial data).
4. The `1` (which was obtained from application of `fact` to `0` in 2.), is a constant in `fact`.

In practice, we would also like to know *which* static occurrence in the program of basic operators is being used, *which* static occurrence of each program constant

is being used, and, where appropriate, the route through function applications that the value has taken from its point of creation to the current use.

We propose

- A *tracing method* that captures this information in a complex data structure.
- A *browser* that allows us to selectively extract explanations such as those above.

The tracing method must satisfy a number of requirements:

1. The trace representation must be (reasonably) easy to browse.
2. In order to scale up well, the execution trace must be fairly compact for large examples. In particular, we would like to avoid functions like `f x = x + x` duplicating `x`'s story for each occurrence of `x` in `x + x`.
3. In order to handle higher order working, the function component of applications also needs to have an attached story. (This is particularly tricky for where curried functions are partially applied, and where 'sections' are used.)
4. The approach must not be over strict. Since we are considering lazy evaluation, in applications where a non-terminating computation (or infinite data structure) is supplied for a non-needed argument, the inclusion of that argument's story in the overall story must not render other parts of the overall story unreachable (and so unreadable). We will exploit the laziness in the debugging scheme to solve this problem.

The next section we describe the details of our tracing scheme, which currently entails *augmentation of the program's source code*. This gives us a fully functional debugger for functional programs.

3 Representing traces: Debug Trees

Fundamentally we are attempting to discover information about each function call. There are three components that we need to keep track of: the function itself, the input arguments to the function and the result of applying the function to its input arguments. The details we are attempting to capture can be represented as two interdependent data types: one for the function and its arguments (application tree, `Atree`) and one for the result of the application (debug tree, `Dtree`). In fact, `Dtree`s capture the general notion of 'story' above.

In Gofer, as in most functional languages, function is 'curried': that is, each function is treated as having only one argument. Higher order functions allow us to construct functions of more than one argument. Consider the function application `f x`, where both `f` and `x` are *expressions*, the former yielding a function value. In our scheme the node constructor `FnArg` will represent the application. The first branch is the function `f` and the second its input argument. We will call this datatype `Atree` for application tree.

```
data Atree  =  FnArg Dtree Dtree
```

We capture the result of the application in our second datatype, `Dtree`, for debug tree. Now we consider the actual structure of `Dtree`. `Dtree` must capture the

result of applications as well as basic types, operators, function names and lists. Below is the definition of `Dtree`:

```
type FName  =  [Char]                 -- Function names
type VName  =  [Char]                 -- Variable names
data Bop    =  Plus | Min | ...| Not  -- Built in operator names

data Dtree  =  Lf Int |               -- Leaf:  integer constant
               Lfb Bool |             -- Leaf:  boolean constant
               Lfbi Bop |             -- Leaf:  built in operators
               Lff FName |            -- Leaf:  function name
               Lfvar VName Dtree |    -- Variable names
               ⋮
               App Atree Dtree |      -- Application and result
               Partial                -- Placeholder for result of a
                                      -- partial application
```

Now we have a rich data structure to capture stories. Note that the ADT includes function names and operators. This will allow us to capture in a generalised way the name of the function being applied as well as the values it was applied to. The leaf for built in operators is not strictly necessary as these can be dealt with via the normal function name leaf. They are left in, at present, so that they can be differentiated from user defined functions—this may or may not prove useful.

We have omitted from the presentation of `Dtree` the elements that allow us to record which static occurrences of constants and operators are used, and which variables are accessed to obtain values. These are easy to add, but would complicate the appearance of the example stories below.

Now we must examine the systematic introduction of debugging trees into programs.

3.1 Constants

Integer constants are represented by the leaf, `Lf`, of the `Dtree`. For instance, 7 must be augmented to `(7, Lf 7)`. We will call the new values, *traced values* and define the type synonym `type T x = (x, Dtree)`. At present only integer and boolean (`Lfb`) constants are defined.

3.2 Variables

Variables are replaced by:

```
(v, Lfvar ``x'' s) where (v, s) = x
```

"x" is the name of the variable in question.

3.3 Built in Operators

Each built in operator, such as multiplication $*$, is replaced by a version which expects traced operands and yields a traced result. For example, we replace each occurrence of a fully applied $*$ such as $e_1 \ * \ e_2$ by `mult` e_1 e_2 where:

```
mult                    ::  T Int -> T Int -> T Int
mult (x, xd) (y, yd)    =   (x * y, Lf (x * y))
```

This definition appears to 'throw away' the traced elements `xd` and `yd` of the operands. The traced components of the operands are dealt with by the 'traced' definition of application itself (see next). Also, note the above is not the full story for expressions involving primitive operators. Further details are in 3.5.

3.4 Application

All functions are treated as curried, and so function application only involves one argument. To deal with the fact that everything is traced we must modify the definition of function application. Usually apply is represented by juxtaposition, i.e f applied to x is normally written fx; we shall represent this with an explicit application operator as `ap` fx, with the implicit definition `ap f x = f x`. When tracing the function application `ap` e_1 e_2, e_1 will yield a traced function, say (f, fd), and e_2 will yield a traced argument value, say, (x, xd). The function f must be applied to (x, xd), yielding result, (x', xd'). Finally the story of the application and how it computes its result must be constructed from fd, xd, xd'. Thus we recast apply as:

```
ap (f, fd) (x, xd)   =  let (x', xd')  =  f (x, xd)
                        in (x', App (FnArg fd xd) xd')
```

For example, applying the traced version of 'not' (`pnot`) to, say *True*, is written as:

```
ap (pnot, Lfbi Not) (True, Lfb True)
```

which yields:

```
(False, App (FnArg (Lfbi Not) (Lfb True)) (Lfb False))
```

3.5 Curried Functions

When dealing with curried functions we must retain the full trace of the applications. Recall that an expression like 2 $*$ 3 represents the application of $*$ to 2 and then the application of the curried function $2*$ to 3. In our notation this would be written as `ap (ap f x) y`. The inner apply (`ap f x`) must return an entity with type `T (T Int -> T Int)`, i.e, a function that takes a traced integer and returns a traced integer and the trace of the application (`ap f x`). The value `(mult (2, Lf 2), Partial)` has the right type but applying this to, `(3, Lf 3)`, yields `(6, App (FnArg Partial (Lf 3)) (Lf 6))` which loses the tracing information about `mult` being applied to `(2, Lf 2)`. To retain this information we must replace the 'Partial' in `(mult (2, Lf 2), Partial)` with the proper trace representing $2*$:

```
App
   (FnArg (Lfbi Mult) (Lf 2))
   Partial
```

This represents the application of the built in operator `mult` to `(2, Lf 2)`, i.e. $2*$. The result of this application is represented by the placeholder leaf `Partial` from `Dtree`. This is the trace representation of `mult (2, Lf 2)`. Thus 2 $*$ 3 can be rewritten as:

```
ap
  (mult (2, Lf 2), App (FnArg (Lfbi Mult) (Lf 2)) Partial)
  (3, Lf 3)
```

This version gives us the full trace information. As can be seen from the results of running the above:

```
(6,
App
  (FnArg (App (FnArg (Lfbi Mult) (Lf 2)) Partial)
              (Lf 3))
  (Lf 6))
```

We can package up the tracing for a fully curried, tracing version of multiplication, `pmult`, so that we can write down expressions like `ap (ap pmult (2, Lf 2)) (3, Lf 3)` to represent `2 * 3`. To do this we define `pmult` as:

```
pmult  ::  Num Int   =>  T (T Int -> T (T Int -> T Int))
pmult  =   let f' x  =   (mult x, Partial)
           in (f', Lfbi Mult)
```

Note that this means we have a way of writing sections. From the above we can see that the section `2*` is rewritten as `ap pmult (2, Lf 2)`.

The above scheme generalises for any function of two arguments `f :: a -> b -> c`. A new function `pf` is defined as:

```
pf  ::  T (T a -> T (T b -> T c))
pf  =   let f' x  =  (f x, Partial)
        in (f', Lff "f")
```

(Note that this calls the *original* `f`)

A function of one argument `f :: a -> b`:

```
pf  ::  T (T a -> T b)
pf  =   (f, Lff "f")
```

A function of three arguments `f :: a -> b -> c -> d` is rewritten as:

```
pf  ::  T (T a -> T (T b -> T (T c -> T d)))
pf  =   let  f' x y  =  (f x y, Partial)
             f' x    =  (f' x, Partial)
        in (f', Lff "f")
```

This scheme generalises easily to functions of n arguments.

In general, an occurrence of the name of a *program defined function* `f` in an *expression* is represented as `pf`, where `pf` is a new function as defined above. Where `f` occurs not in an application expression (e.g. as argument to a higher order function), it is left as `pf`. All applications are transformed into fully curried form using `ap`:

`f` e_1	is replaced by	`ap pf` e_1'
`f` e_1 e_2	is replaced by	`ap (ap pf` e_1'`)` e_2'
`f` e_1 e_2 e_3	is replaced by	`ap (ap (ap pf` e_1'`)` e_2'`)` e_3'
$\vdots$	$\vdots$	$\vdots$

where e_n' is the appropriate form of e_n.

An occurrence in an expression of a variable representing a function valued argument needs no special treatment, as its value will already be properly traced.

3.6 Function definitions

Functions are defined as collections of equations of the form

```
f x y ... = e
```

where the arguments `x`, `y`, ...may contain patterns.

The function definition must be transformed by replacement of all constants c in the patterns by `(`c`, cd)` (which will result in the stories associated with such arguments being discarded, as they do not contribute to the result), and by applying the augmentations described above to the body of the function.

Actually required is an indexing of the equations to aid back tracing and similiarly with guards (not discussed here).

3.7 An Example

Returning to the factorial example. We rewrite `fact` and introduce a new function `pfact`:

```
fact (0, nd)   =  (1, Lf 1)
fact n         =  ap (ap pmult (v, Lfvar "n" s))
                     (ap pfact (ap (ap pminus (v, Lfvar "n" s)) (1, Lf 1)))
                     where (v, s) = n
pfact          =  (fact, Lff "Fact")
```

Running `ap pfact (1, Lf 1)` (and prettifying a little by hand) gives:

```
App
  FnArg (Lff Fact) (Lf 1)
  App
    FnArg
      App
        FnArg (Lfbi Mult ) (Lfvar "n" Lf 1)
        Partial
      App
        FnArg
          Lff Fact
          App
            FnArg
              App
                FnArg (Lfbi Min ) (Lfvar "n" Lf 1)
                Partial
              Lf 1
            Lf 0
        Lf 1
    Lf 1
```

We have the result `1` and its trace. The trace is rather hard to read in this form,

but it is a well structured object. It is clear that this method is not easily used without a browser (see 4).

3.8 Infinite Lists and Non-termination

We have not discussed how to trace the computation of list structures, as that is not the main thrust of this paper. However, there is one difficult problem that we should mention: infinite lists. We may call functions with arguments of the form `[1..]`. The traced form of an infinite list contains an infinite `Dtree`. We cannot simply print this out as the input to some function, and we have to be clear that it is safe to allow the infinite `Dtree` to exist in the computation at all!

A similar consideration applies to non-terminating computations that return no value at all, e.g. an application of `f x = f (x + 1)`. The computation will expand to be an infinite `Dtree`.

The solution to both these problems is the same: the `Dtree` is lazily evaluated, and a browser can be directed to show an interactively determined number of the computation steps, thus limiting the debug tree traversed.

Full details of the transformation scheme are given in [2]. This shows all languages features that can be dealt with by our scheme at present except for translation of variables as this is a recent addition to the scheme. Other issues that we must add to the scheme are algebraic datatypes and a mechanism to deal with computations of the form: `1/0`.

4 The Browser

The debug tree for the traced evaluation of `fact 1` in 3.7 above shows that the debug tree itself would make a very poor tool for analysing program behaviour. It is quite hard to follow, even for a computation as simple as `fact 1`. The debug tree for `fact 2` is rather larger, and almost intractable. Therefore we propose to use a *browser tool* for inspection of the debug tree produced by a faulty computation that we wish to investigate. The browser fulfils several important roles:

- It allows us to view only a part of the debug tree at any time, and to gradually and systematically explore its subtrees.
- It presents the contents of the debug tree to us in a more palatable form.
- Since the debug tree is initially in an unevaluated form, due to the lazy evaluation, the browser can control the extent to which we force evaluation of components of the tree.

We have a prototype browser that allows us to interactively explore debug trees of the kind shown in this paper. It incorporates various debugging aids additional to the basic `Dtree` treatment discussed above: `App` nodes record an index of which equation was used when the function was applied (used when displaying the application node); `Dtree` nodes record the *location in the source program* of the construct evaluated (so that, for example, the instance of a constant referred to can be identified accurately); the text of the program being debugged is available, and lines identified by the *locations* referred to above are output to aid the user. We still need to explore fully the capabilities of the debugging browser and to assess what additional practical facilities are required; it also seems to be an excellent candidate for a point-and-click GUI.

4.1 Example

We can use our browser to perform the kinds of investigations illustrated below. The example omits some of the "additional aids" mentioned above, to simplify the presentation.

Let us suppose that we are debugging a faulty version of the `fact` function, as above. The fault that has crept in is that the first equation in the original form of the code has carelessly been entered as

```
fact 0 = 0
```

and that we have evaluated `fact 2` and obtained the result `0`.

Let's assume that we understand the intended computation fairly well, but that the text of the program is sufficiently long and complex that we need help in exploring which parts have been used by the particular evaluation. [Clearly this is not true of `fact`, but it is a common debugging scenario.] We need to find out *why* the result is `0`.

At each step, the browser outputs a summary of the 'top level' of the `Dtree` being viewed, and invites the user to select a sub-component to explore further. The browser attempts to simplify subtrees to their actual values — these are indicated by a `*` suffix below, and are candidates for sub-component exploration. This automatic 'chasing' of a value is restricted in the depth of its search in the `Dtree` (to avoid infinite recursions and 'black holes').

The debug session for `fact 2` starts by applying the browser to the `Dtree` from evaluating `ap pfact (2,(Lf 2))`, and proceeds as follows:

```
Current: Application of fact* to 2* with result 0*

Action? Follow result

Current: Application of Mult 2* to 0* with result 0*

Action? Follow argument

Current: Application of fact* to 1* with result 0*

Action? Follow result

Current: Application of Mult 1* to 0* with result 0*

Action? Follow argument

Current: Application of fact* to 0* with result 0*

Action? Follow result

Current: Constant 0            (in the first equation for fact)
```

At this point the user has found the *source* of the `0`, and can see how it appears as an operand to the multiplications. From an understanding of the intended algorithm, it should be `1`, and so we can consider the debugging investigation at an end. The browser also allows the user to *backtrack* in the investigation, so that should no problem be found in one subtree, another can be explored.

5 Related Work

There has been some research into functional debuggers but it is not intention our to review that work here as our focus is on the proposal presented in this paper. The interested reader is referred to [3], [7], [9], [10], [14].

A debugger for the ML functional language [13] also allows the programmer to work backwards through the program. It does this by taking snapshots of the state of the program at various points (e.g. before function application). Therefore it is possible for the programmer to work backwards from where the error occurred but this debugger does not record the actual details of the calculations that generated the value(s) nor does it attempt to support the 'time-travel' focused on the pinpointed variables. It allows the programmer to work back from the state at 'time' n to 'time' $n - 1$ and examine the value of each variable. This functional debugger is certainly the most complete implementation of a debugger for a functional programming languages—it supports 'traditional' debugging features such as breakpoints and watches—and certainly a useful tool to debug ML programs. The methods used in this debugger cannot be applied to the pure functional language because they exploit non-pure features.

Sparud [11, 12] has also investigated a similiar approach to ours using an Evaluation Dependence Tree (EDT) which is similar to our `Dtree`. The EDT is constructed by 'modifying' apply in a similar way to our mechanism. To support the modification of apply programs are also rewritten (just like in our scheme). We believe that Sparud's scheme can also be used in the same fashion as ours and are currently investigating this claim. One feature of note is that Sparud scheme exploits a special feature of the Chalmers Haskell compiler: *existential types* not supported in the Glasgow implementation of Haskell or in Gofer.

To implement this scheme in the imperative domain it is clear that the use of "Dynamic Slicing" [15, 8, 1] would be of great benefit. Dynamic slicing is a technique that identifies those parts of a program that are involved in the calculation of a particular variable. The slice is guaranteed to have the same behaviour as the whole program with respect to to the variable in question. To implement our scheme we will need to know the data-dependencies of the variable in we wish to investigate and the work done to implement dynamic slicing will clearly be crucial.

6 Conclusions

What we have done so far is sketch out what appears to be a promising approach to the debugging of functional languages and possibly imperative languages.

It is our intention to investigate this approach in more detail to get a firmer grip on the difficulties that stand in the way of full implementation, particularly in the functional domain—certainly we believe we can implement this idea of 'debugging by dataflow' or 'back tracing' with Haskell [4] by rewriting the *standard prelude* in the appropriate form. Then a program could be used to transform a *normal* program into a *debug* version which would then be interactively executed. At present we are investigating doing this in a interpreted language, such as Gofer [5, 6], simply by replacement of the *standard prelude* by a *debugging prelude*. Full details of this work can be found in [2].

The main idea we wish to deal with is not the *functional* implementation of 'debugging by dataflow' but the facility of the proposal—the notion that there is value in capturing the 'story' of how values inspected when debugging have been

constructed during the program's execution. We believe that this combined with a broswer that allowed the programmer to walk 'backwards' through a value's construction provides an alternative to 'normal' (tracing, watches and breakpoints) debugging techniques.

This paper has drawn heavily from the domain of functional programming which simply reflects the authors particular programming basis and where we first used the proposal outlined in this paper to investigate whether it formed the basis of a useful means to help debug functional programs. Recently, though we have began to look at whether it has utility beyond the world of functional programming and although our initial investigation consists of 'thought experiments' we firmly believe that the proposal forms basis of what should be valuable research into how to extend the idea into other programming domains.

Clearly, there will be problems that we have not for seen although memory usage is one that springs immediately to mind. This potential memory problem will not be as bad as it might first appear because the `Dtree` will be evaluated lazily. This is an area that we must investigate in the future so that we can determine the overheads associated with our proposed debugging scheme.

There is also a risk that this proposal would overwhelm the programmer with enormous amounts of information (a criticism often levelled with justification at techniques like algorithmic debugging which probably present less information to the programmer). This will be some extend be alleviated by the fact that the programmer is driving the process just not mindlessly being presented with information—this would certainly be an area for refinement.

Overall, though, we believe that the ability to 'trace back' through the ancestry of a particular value is a powerful debugging technique that would be of use to programmers actually debugging programs.

References

[1] H Agrawal and JR Hogan. Dynamic program slicing. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] SP Booth and SB Jones. Towards a purely functional debugger for functional programs. In *Functional Programming, Glasgow 1995*, pages 1–12. Springer-Verlag, electronic Workshops in Computing, 1996. Also Technical Report CSM139. Department of Computing and Mathematics, University of Stirling.

[3] CV Hall and JT O'Donnell. Debugging in a side effect free programming environment. In *ACM Sigplan 85 Symposium on Languages Issues in Programming Environments*, pages 60–68, June 1985.

[4] P Hudak, S Peyton Jones, and P Wadler et al. *Report on the Programming Language Haskell*, March 1992.

[5] MP Jones. *Gofer 2.28 release notes*, February 1993. Available as part of the Gofer distribution `ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer`.

[6] MP Jones. *An introduction to Gofer*, 1994. Available as part of the Gofer distribution `ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer`.

[7] A Kishon and P Hudak. Semantics directed program execution monitoring. *jpf*, 5(4):501–547, Oct 1995.

[8] N Shahmehri M Kamkar and P Fritzson. Interprocedural dynamic slicing. Technical Report LiTH-IDA-R-91-20, Department of Computer and Information Science, Linköping Univerity, 1991.

[9] L Naish. Declarative debugging of lazy functional programs. Technical report, University of Melbourne, June 1992.

[10] L Naish. A declarative debugging scheme. Technical report, University of Melbourne, January 1995.

[11] J Sparud. Towards a Haskell debugger. Chalmers University, `http://www.cs.chalmers.se/~sparud/`.

[12] J Sparud and H Nilsson. The architecture of a debugger for lazy functional languages. In *AADEBUG'95, 2nd International Workshop on Automated and Algorithmic Debugging*, May 22-24 1995.

[13] AP Tolmach and AW Appel. A debugger for standard ML. *jpf*, 5(2):155–200, 1995.

[14] I Toyn. *Exploratory Environments for Functional Programming.* Ph.d thesis, University of York, April 1987.

[15] M Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.