J. Functional Programming **4**(3): 337–370, July 1994.

Algorithmic Debugging for Lazy Functional Languages[1]

Henrik Nilsson and Peter Fritzson

Programming Environments Laboratory
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
E-mail: henni@ida.liu.se, paf@ida.liu.se

Abstract. Lazy functional languages have non-strict semantics and are purely declarative, i.e. they support the notion of referential transparency and are devoid of side-effects. Traditional debugging techniques are, however, not suited for lazy functional languages since, computations generally do not take place in the order one might expect. Since *algorithmic debugging* allows the user to concentrate on the declarative aspects of program semantics, and will semi-automatically find functions containing bugs, we propose to use this technique for debugging lazy functional programs. Because of the non-strict semantics of lazy functional languages, arguments to functions are in general partially evaluated expressions. The user is, however, usually more concerned with the values that these expressions represent. We address this problem by providing the user with a *strictified* view of the execution trace whenever possible. In this paper, we present an algorithmic debugger for a lazy functional language based on strictification and some experience in using it. A number of problems with the current implementation of the debugger (e.g. too large trace size and too many questions asked) are also discussed and some techniques for overcoming these problems, at least partially, are suggested, the key techniques being *immediate strictification* and *piecemeal tracing*.

1 Introduction

Debugging has always been a costly part of software development, and several attempts have been made to provide automatic computer support for this task [Sev87]. The algorithmic debugging technique introduced by Shapiro [Sha82] was the first attempt to lay a theoretical framework for program debugging, and to take this as a basis for a partly automatic debugger.

Algorithmic debugging is a two phase process: an execution trace tree is built at the procedure/function level during the first (trace) phase. This tree is then traversed during the second (debugging) phase. Each node in the tree corresponds to an invocation of a procedure or function and holds a record of supplied arguments and returned results. Once built, the debugger basically traverses the tree in a top-down manner, asking, for each encountered node, whether the recorded procedure or function invocation is correct or not. If not, the debugger will continue with the child nodes, otherwise with the next sibling. A bug has been found when an erroneous application node is identified where all children (if any) behaved correctly.

Algorithmic debugging was first developed in the context of Prolog. In previous research by our group, the algorithmic debugging method has been generalised to a class of imperative languages and its bug finding properties improved by integrating the method with program slicing [Sha91][FGKS91][KSF92][Kam93].

Within the field of lazy functional programming, the lack of suitable debugging tools has been apparent for quite some time [Aug91]. As has been pointed out earlier by others [HO85]

1. This work has been supported by the Swedish National Board for Industrial and Technical Development (NUTEK).

[OH88][TR86][Toy87], we feel that traditional debugging techniques (e.g. breakpoints, tracing, variable watching, etc. in the *conventional* sense) are not particularly well suited for the class of lazy functional languages, since computations in a program generally do not take place in the order one might expect from reading the source code.

Algorithmic debugging, however, allows a user to concentrate on the declarative semantics of an application program, rather than its operational aspects such as evaluation order. During debugging, the user only has to decide whether a particular function applied to some specific arguments yields a correct result. Given correct answers from the user, the debugger will determine which function that contains the bug. Thus, the user need not worry about why and when a function is invoked, which suggests that algorithmic debugging might be a suitable basis for a debugging tool for lazy functional languages.

Obviously, there must be an *externally visible bug symptom* for this technique to work: e.g. if two bugs conspire to hide each other, it will not be possible to (algorithmically) debug the program. But if there is no bug symptom, the user is unlikely to undertake any debugging in the first place, so this is not really any major drawback.

To test the idea in practice, support for algorithmic debugging was added to an existing compiler for a small lazy functional language and an algorithmic debugger, LADT (Lazy Algorithmic Debugging Tool), was implemented. The language, called Freja [Nil91], is essentially a subset of Miranda[1] [Tur85]. It is based on graph reduction and implemented using a G-machine approach [Aug84][Aug87][Joh84][Joh87][Jon87][2]. LADT was successfully applied to a number of small examples, thus showing the relevance of algorithmic debugging for lazy functional programming.

However, as we gained experience from using this early system, a number of problems became apparent. First, the questions asked were often big and complex, involving large data structures and unevaluated expressions. Second, storing the complete trace is impractical for any but the smallest of problems. Finally, the number of questions asked by the debugger for realistic programs is too large.

To alleviate the first problem, a technique which we term *strictification* was introduced. This technique is concerned with hiding the lazy evaluation order, thus giving the user an as strict impression of lazy execution as possible. This reduces the number of unevaluated expressions involved in questions posed to the user, which, in our opinion, tends to make them easier to understand and answer. Note that it is not possible to simply evaluate unevaluated expressions as they are encountered during the debugging phase, since they may represent infinite structures or non-terminating computations.

Our current implementation thus does strictification. This leaves the last two, rather severe, problems. We do not believe that an algorithmic debugger will be practically usable unless they are addressed in a satisfactory manner. A number of possible approaches for overcoming these problems, at least to some extent, are therefore outlined in this paper after the description of the current system.

The rest of this paper is organised as follows. In section 2 the basic problems of debugging lazy functional programs are reviewed. The principles behind algorithmic debugging are then explained in section 3. Section 4 describes how algorithmic debugging may be adapted for lazy functional programs and develops the idea of *strictification*. Section 5 gives some details on the current implementation of LADT, which is then evaluated in section 6. A few ideas for practical

1. Miranda is a trademark of Research Software Ltd.
2. In comparison with some other compilers for lazy functional languages, this compiler is rather basic; in particular, it does not do any strictness analysis and it does not perform fully-lazy lambda-lifting.

implementation are presented in section 7, and related work then discussed in section 8. Finally some conclusions are given in section 9.

2 The Need for Lazy Debugging Tools

Consider the following functional program, where `foo` is a function that clearly must not be applied to the empty list:

```
foo xs = 0, if hd xs < 0
       = hd xs, otherwise;

fie xs = (foo xs):fie (tl xs);

main = fie [-1];
```

The problem is that there is no termination condition in the recursive function `fie`, which means that `fie` eventually will apply `foo` to the empty list since `main` applies `fie` to the *finite* list `[-1]`. Suppose that we have strict semantics. Then we would get an execution trace tree as shown in figure 1.

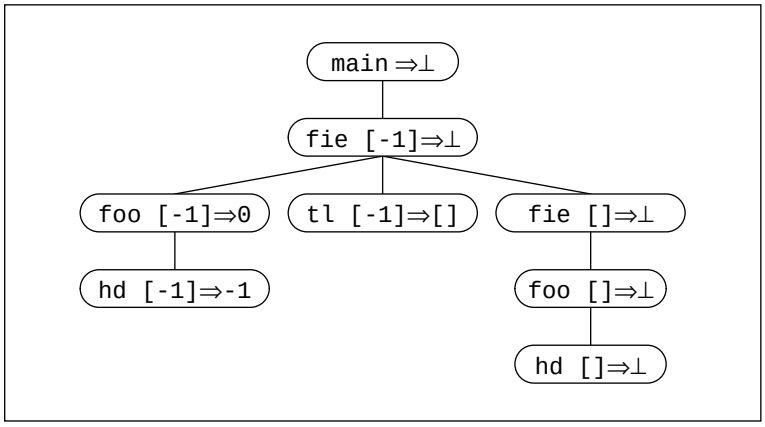

Fig. 1. Strict execution trace tree after run-time error.

In the rightmost branch, we see how the application of `hd` to the empty list provoked a runtime error, represented by the symbol ⊥ (“bottom”). At this point, we simply have to follow the edges from the leaf node towards the root, which in practice is easily achieved by inspecting the runtime *call* stack, to find out that the problem is that `fie` has applied `foo` to `[]`.

Now suppose that we had had lazy semantics instead. We would then get an execution tree as shown in figure 2. Applying the same technique as above to this tree would not give any insight as to what the problem might be: `foo` is applied to an expression that will evaluate to the empty list, that is for sure, but which function applied `foo` to that expression? There is no recollection of this in any of the execution tree nodes on the path from the node that caused the error to the root node, which are the only nodes available to a conventional debugger in the form of the runtime stack. The key difference is that we now have a *demand* stack rather than a call stack.

The presence of (partially) evaluated expressions in the lazy execution tree should also be noted. In general, these may become very large and complicated, even if they only denote very simple values, thus making it even harder for a user to get a clear understanding of what is going on.

We could also try to do a conventional trace of the lazy execution:

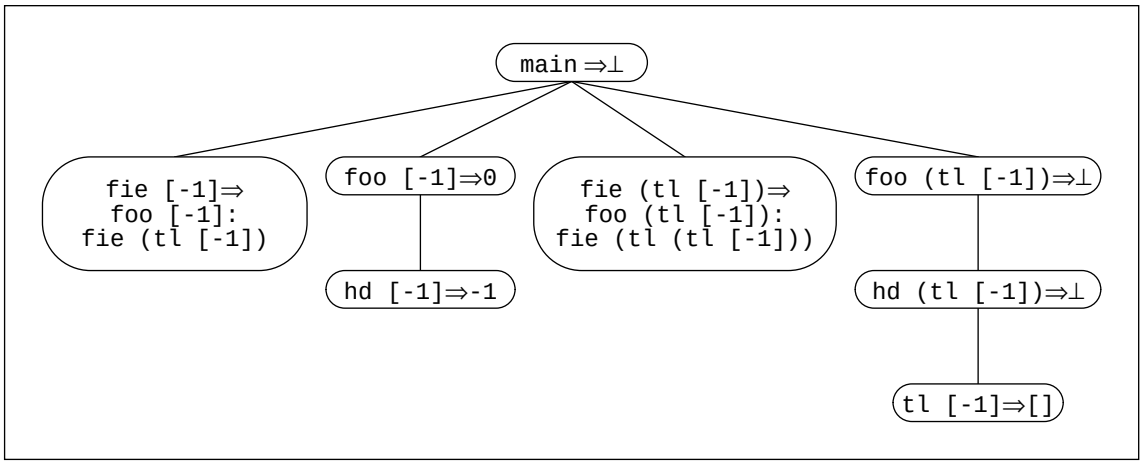

Fig. 2. Lazy execution trace tree after run-time error.

```
Entering fie:
     xs = [-1]

Leaving fie, result is:
     (foo [-1]):(fie (tl [-1]))

Entering foo:
     xs = [-1]

...
```

Apparently, `fie` did not call `foo`! This behaviour is probably not in agreement with the programmer's mental model of the execution process, and it is at least not obvious from reading the source code. In any case, lazy evaluation permits the programmer to largely forget about evaluation order, so there is a strong case for not bothering him with it during debugging.

To sum up, debugging of lazy functional programs is made difficult by delayed evaluation, counter-intuitive evaluation order and partially evaluated expressions. Because of this, traditional debugging tools and techniques are of little or no use. Hence "lazy" debugging tools, that address the above mentioned problems and that conceptually fit into a lazy framework, are needed, even though these might still be based on conventional ideas (e.g. tracing) (see also section 8).

3 Basic Algorithmic Debugging

In this section we describe the basic principles of algorithmic debugging. To simplify the presentation and provide a background to our work, we use a debugger based on the original algorithmic program debugging method by Shapiro [Sha82].

Algorithmic debugging is based on the notion of an *externally visible bug symptom*, i.e. an execution of a program did not produce the expected result. This must have a cause: either the topmost procedure did something wrong, or some procedure invoked from the topmost one produced some erroneous result. By inspecting the invocations performed from the topmost procedure, it is possible to determine what is the case: if all invocations yielded correct results, then the topmost procedure must be at fault. Otherwise, one of the invocations exhibits some visible bug symptom, and we may thus apply the above reasoning over again, tracing the source of the bug down the execution trace tree. Since the debugger cannot know what is a bug

symptom and what is correct behaviour, at least not initially, there must be an *oracle*, i.e. the user, ready to supply it with this information.

Given a visible bug symptom, algorithmic debugging guarantees that the procedure or function containing the bug will eventually be found, provided that the user answers the questions about the program behaviour correctly. If there is more than one bug in a program, only one of them will be found. However, algorithmic debugging may be applied again, once this bug is removed, in order to find the next bug.

In more detail, algorithmic debugging proceeds as follows. The debugger first executes the program and builds an execution trace tree at the procedure level. A node in the tree is constructed for each procedure invocation and essential trace information, such as the procedure name and the values of all input and output parameters, are recorded. If any further procedure calls are made during the current invocation, these become children of the current node. New children are inserted from left to right as the corresponding procedures are invoked.

Once the execution has finished, the algorithmic debugger starts searching for the bug by traversing the execution tree in a preorder manner. For each node, the debugger interacts with the user by asking whether or not the behaviour of the procedure invocation corresponding to the node is correct. The user may reply yes or no to this[1]. If a positive answer is given, the search continues from the next branch to the right of the current one, otherwise the search continues from the leftmost child of the current node (if it has any). The search ends at a node exhibiting incorrect behaviour when one of the following two conditions holds:

- No further procedure calls were made during the procedure invocation corresponding to this node; i.e. this node has no children.
- All procedure calls performed during the procedure invocation corresponding to this node fulfil the user's expectations.

Finally, the debugger reports the name of the offending procedure. The user's answers are remembered in a database (which could be preserved across debugging sessions) so that the same question does not have to be asked twice.

Figure 3 shows an example of algorithmic debugging. The bug is in the function `insert`. Since the user answers no to the first two questions, the debugger moves down to the leftmost child of `sort([1,3])`. According to the user, the behaviour recorded in this node is correct, so the next branch to the right is investigated next. The behaviour of that node is wrong, but its only child behaved correctly. The bug must therefore be in the body of the procedure that is associated with this node, i.e. in `insert`.

Note that, to be able to give a correct answer, the user must be given the "full picture" for each question, i.e. the computation must depend only on its input parameters, and there must be no effects of the computation besides what is indicated by the output parameters. Procedures that fulfil this criterion are said to be *side-effect-free*. It is possible to achieve side-effect-freeness in programs written in imperative languages by program transformations [Sha91].

Note also that programs that abort due to runtime errors as well as programs that loop, easily can be handled within the algorithmic debugging framework (in the latter case provided that the user interrupts the program) if procedures and functions conceptually are allowed to return bottom, $\perp$, representing a diverging computation.

1. Shapiro's original method [Sha82] could only handle yes/no answers. A capability of handling assertions about the intended behaviour of a procedure was presented later by Drabent, Nadjm-Tehrani and Maluszynski [DNTM88], and is also present in the design of GADT.

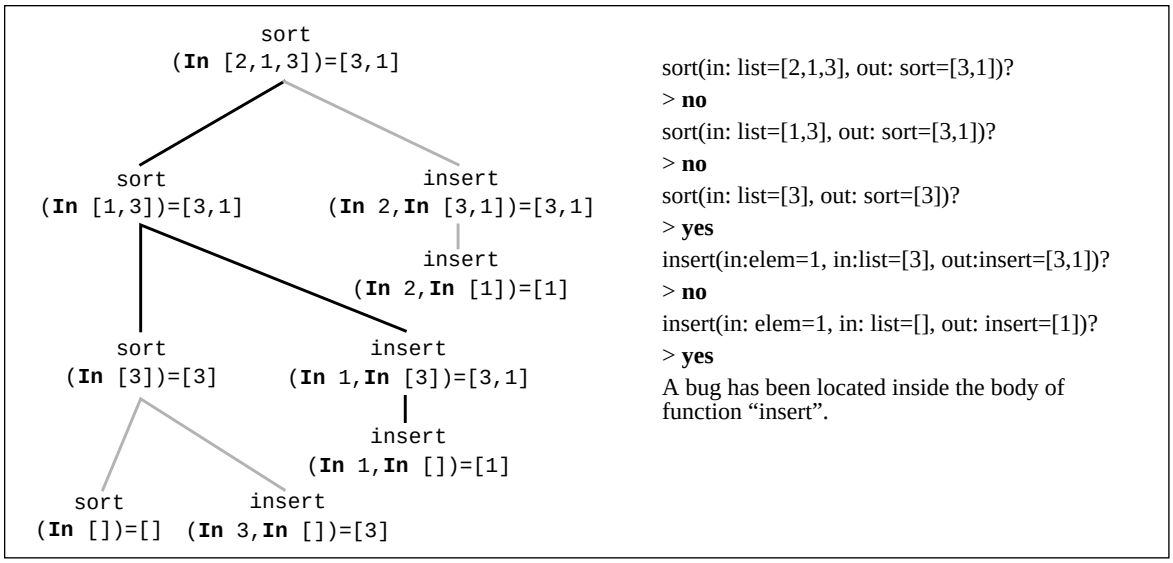

Fig. 3. The execution tree of an erroneous sort program and the ensuing user interaction.

4 Lazy Algorithmic Debugging

In this section, we describe how algorithmic debugging may be applied to programs written in lazy functional languages. In section 4.1, we explain how to use the basic algorithm on this domain. Then, in sections 4.2 and 4.3, strictification is developed as a means for making algorithmic debugging of programs written in lazy functional programs easier. The resulting debugging algorithm is then presented in section 4.4. Finally, in section 4.5, we provide a small example of algorithmic debugging of a functional program.

4.1 The Basic Approach

Basic algorithmic debugging [Sha82] may readily be used for lazy functional languages since functions in such languages *are* side-effect-free. We do have to regard the execution environment of a function as belonging to its input parameters, though, since functions may reference *free variables*.

However, while gaining experience in using an early version of the debugger, the fact that arguments to functions in general are partially evaluated expressions soon proved to be a major problem: the user is usually concerned with the values that these expressions represent, whereas details of the inner workings of the underlying evaluation machinery often are of little or no interest. Furthermore, the questions that the debugger asked were frequently textually very large and difficult to interpret.

This problem might have been especially apparent in our case since our compiler does not do strictness analysis, which is something any good compiler would do. However, strictness analysis is only an approximation, and there will always be occasions when unevaluated expressions are passed as parameters but used later (otherwise we would not need the lazy semantics in the first place).

This suggests that we should replace unevaluated expressions by the values they represent, wherever possible, to give the user an impression of strict evaluation, which probably is closer

to the user's mental model of the evaluation process anyway. We will refer to the technique of giving an impression of strict evaluation (if possible) as *strictification* from now on.

However, strictification is, as illustrated in the following section, not as straightforward as it first might appear: not only must values be substituted for expressions wherever possible, but the actual structure of the execution tree has to be changed as well. Otherwise, there is no longer any guarantee that the bug will be found.

4.2 Why Substitution of Values for Expressions Is Not Enough

Suppose that strictification was implemented in the obvious way, i.e. before asking whether a function application yielded a correct result or not, any unevaluated expressions that occur in the arguments or in the result of the application are replaced by the results of evaluating the expressions in case they are known to the debugger (i.e. were needed at some point during the execution)[1]. This, of course, has to be done recursively should the results themselves contain any unevaluated subexpressions. The user will then see a version of the function application which is as strict as possible given a particular execution trace.

Unfortunately, the debugging algorithm is then no longer guaranteed to find the bug, as illustrated below (the function `add` is incorrect):

```
dbl x = add x x;
add x y = x * y;
main = dbl 3;
```

If no strictification is performed, evaluating main would yield an execution tree as depicted in figure 4. The debugger quickly concludes that the bug must be in the function `add` since applying `add` to `3` and `3` yields something erroneous and since that node has no children.

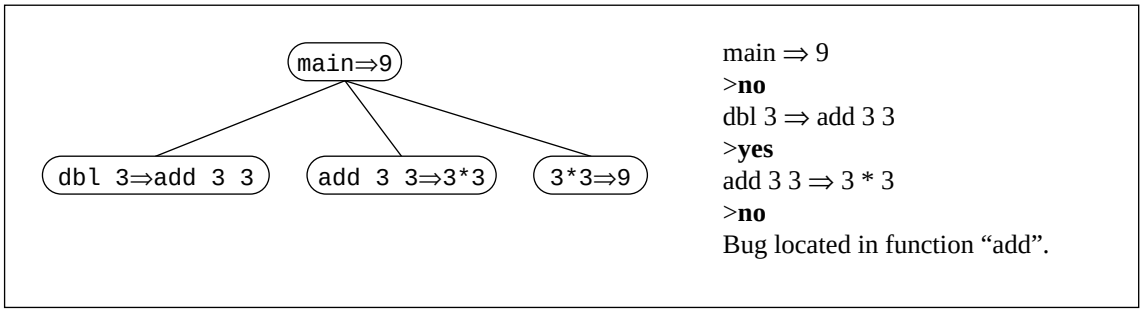

Fig. 4. Lazy execution trace tree.

Now, suppose that we did substitute values for expression wherever possible in the tree above and then tried to do algorithmic debugging. The result is shown in figure 5. When asked whether `main` should evaluate to `9` the user answers no and the debugger proceeds to the first child node and asks whether `dbl 3` should evaluate to `9` or not. Since the intention is that `dbl` should double its argument the user again answers no. Now, since this node has no children, the debugger will come to the conclusion that the bug is within the function `dbl`, which is wrong.

1. This could also be implemented in a more efficient way by keeping pointers from the execution tree to the graph representation of the unevaluated expressions. Due to the nature of graph reduction, these pointers would at the end of the execution refer to graphs representing the expressions in their most evaluated form, which is the most we can hope for under lazy evaluation unless we are going to change the semantics of the language by forcing further evaluation, something a debugger should not do and that could change a terminating program into a non-terminating one.

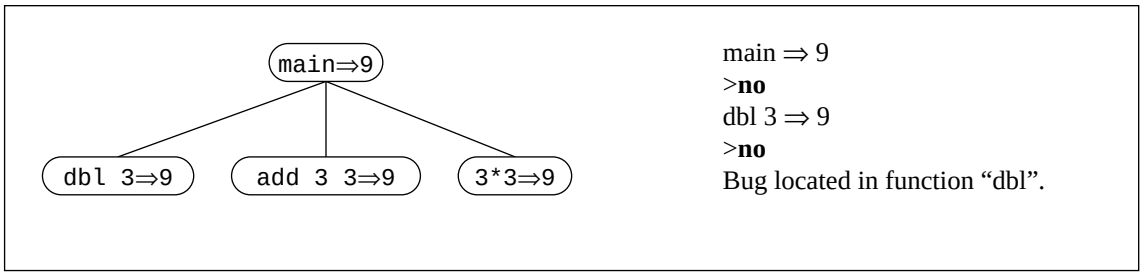

Fig. 5. Incorrectly strictified execution trace tree.

The problem is that in doing the substitutions (in this case first substituting `3*3` for `add 3 3` and then `9` for `3*3`) we are effectively pretending that these computations take place at an earlier point in time than is actually the case, but this is not reflected in structure of the execution tree. A correctly strictified tree and the resulting debugging interaction may be seen in figure 6. Note how the nodes involved in the substitution have become child and grand child of the `dbl 3` node.

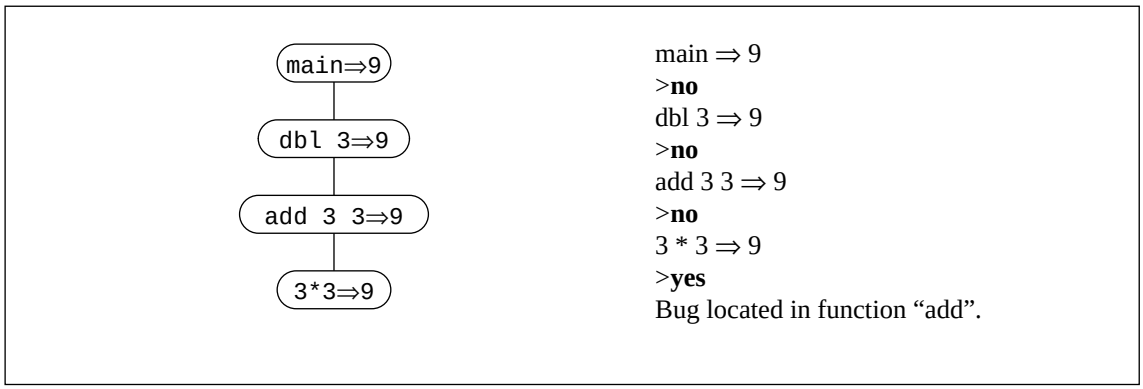

Fig. 6. Correctly strictified execution trace tree.

4.3 Correct Strictification

A correct way of doing strictification can be derived from the following two observations:

- Performing a substitution of the result of evaluating an expression for the expression in one of the arguments of a function application, corresponds to evaluation of the expression in question *before* entering the function (precisely what happens in a strict language). Thus, a new node should be inserted in the execution tree to the left of the node corresponding to the function application.
- Performing a substitution of a result for an expression in the result of a function application, corresponds to evaluation of the expression in question *during* the invocation of the function (again as in a strict language). Thus, the node corresponding to the function application should be given a new child; inserting it to the right of its siblings will do.

These transformations should be applied recursively, i.e. the new nodes must themselves be strictified. The process is depicted in figure 7 for a simple case. Grey nodes represent computations that have taken place elsewhere, `e1`, `e2` and `e3` are expressions.

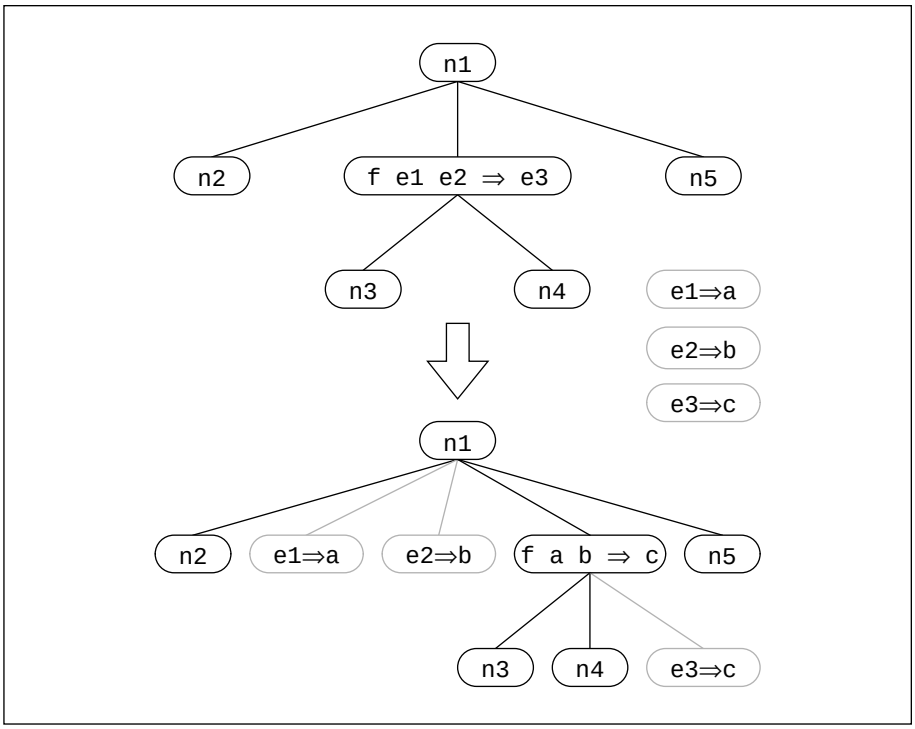

Fig. 7. Correct strictification.

However, there is no need to actually perform these transformations on the execution tree; it is sufficient to ask the questions during the debugging phase in such an order as if the transformation had been performed. This is how strictification is presently implemented in LADT.

4.4 Algorithmic Debugging with Strictification

The algorithm for algorithmic debugging with strictification is given in pseudo code in figure 8. The implemented debugger, in addition to yes and no answers, also supports “maybe” answers and a kind of simple assertions, details of which are omitted from the pseudo code for reasons of simplicity. These features are instead outlined informally.

Also missing from the pseudo code are checks for preventing attempts to strictify nodes in the execution tree that already are being strictified. This can be achieved by keeping nodes currently under strictification in a list. Then, before strictification of a new node is started, it is checked whether it is already present in the list or not. If it is, it can safely be ignored. Similar arrangements must be provided for the evaluation procedures below.

Two abstract data types are assumed: `exec_tree` and `expr`. An execution tree node consists of a function application and its result (both of type `expr`) and zero, one or more children. These are accessed by means of the functions `App`, `Res` and `Children`. A function `Tree` that returns the execution tree corresponding to a function application is also supposed to exist. This function can, of course, only be used if the application was evaluated during the execution, which is checked by the function `Has_Been_Evaluated`.

An expression is either an application of a function to some arguments; a constructed data object (e.g. a tuple) having zero, one or more fields; or some atomic object such as an integer. The functions `Args` and `Fields` are used to access the arguments of an application and the fields of a constructed object, respectively.

There is also a database that can be queried about the status of a function application. Initially, the status is “unknown”. This is changed to “correct” whenever the user answers that

```
(* Locates the bug in the execution tree. *)
PROCEDURE Strictify_And_Debug(tree: exec_tree)
VAR strictified_app, ev_res: expr;
BEGIN
   IF Status_Is_Unknown(App(tree)) THEN
      strictified_app :=
         Make_App(Fun_Name(App(tree)),
                  MAP Evaluate_And_Debug Args(App(tree)));
      ev_res := Evaluate(Res(tree));
      Pretty_Print(strictified_app); PRINT "=>";
      Pretty_Print(ev_res); PRINT "? ";
      IF User_Answer = "YES" THEN
         Set_Status_To_Correct(App(tree));
      ELSE
         FOR ALL c IN Children(tree) DO
            Strictify_And_Debug(c);
         ENDFOR;
         Evaluate_And_Debug(Res(tree));
         (* If we get here, no bug in children. *)
         PRINT "Bug located in function ";
         PRINT Fun_Name(App(Tree)); PRINT ".\n";
         EXIT; (* Exit immediately when bug found. *)
      ENDIF;
   ENDIF;
   (* If we get here, the behaviour was correct *)
END;

FUNCTION Evaluate_And_Debug(e: expr): expr
BEGIN
   IF Is_App(e) THEN        (* Function application *)
      IF (Has_Been_Evaluated(e)) THEN
         Strictify_And_Debug(Tree(e));
         RETURN Evaluate(Res(Tree(e)));
      ELSE
         RETURN Make_App(Fun_Name(e),
                         MAP Evaluate_And_Debug Args(e));
      ENDIF;
   ELSEIF Is_Constr(e) THEN (* Tuple, CONS-cell... *)
      RETURN Make_Constr(Constr_Name(e),
                         MAP Evaluate_And_Debug Fields(e));
   ELSE                        (* Integer, character...*)
      RETURN e;                (* Already fully evaluated. *)
   ENDIF;
END;

FUNCTION Evaluate(e: expr): expr
BEGIN
   IF Is_App(e) THEN        (* Function application *)
      IF (Has_Been_Evaluated(e)) THEN
         RETURN Evaluate(Res(Tree(e)));
      ELSE
         RETURN Make_App(Fun_Name(e),MAP Evaluate Args(e));
      ENDIF;
   ELSEIF Is_Constr(e) THEN (* Tuple, CONS-cell... *)
      RETURN Make_Constr(Constr_Name(e),
                         MAP Evaluate Fields(e));
   ELSE                        (* Integer, character...*)
      RETURN e;                (* Already fully evaluated. *)
   ENDIF;
END;
```

Fig. 8. Algorithmic debugging with strictification.

an application behaved correctly. Thus the same question need not be asked twice. The remaining function and procedure names should hopefully be self-explanatory.

There are, of course, cases when the user might be unwilling to supply a definite yes or no answer to a question, either because he really does not know the answer, or because the question seems to be large and complicated. Thus, there is an option to answer "*maybe* this is correct, I don't know". This effectively lets the user postpone answering the question and get on with the debugging in the hope that a bug is positively identified before he is faced with this question again.

Initially, a "maybe" answer is treated as "no". Thus the children of the current node will be searched. If no bug is found, the algorithm will eventually get back to the node that the user was unsure about, knowing that all its children behaved correctly. At this point, the user is again asked about the node. If he this time indicates that the result is correct, debugging will proceed as usual, the only difference being that the user has had to answer more questions than otherwise would have been the case. If the answer is that the result is incorrect, then there must be a bug in the applied function. However, if the users insists on answering "maybe", the algorithm is forced to conclude that there might be a bug in the function applied. This is reported to the user and the algorithm then continues as if the result was correct, since it is still possible that a more definite bug will be found.

The assertion facility gives a user the possibility to suppress questions regarding a particular function (i.e. asserting its correctness). This has proved to be a quite convenient feature, but obviously it is very primitive as far as assertions go.

4.5 Algorithmic Debugging of a Small Program

The following Freja program, that is supposed to calculate the first five prime numbers, contains a bug in the function `not_div_x` defined locally in `sieve`; the operator equal (`==`) should be replaced with the operator not equal (`~=`):

```
shownums [] = [];
shownums (x:xs) = shownum x ++ " " ++ shownums xs;

filter p [] = [];
filter p (x:xs) = x : filter p xs, if p x
                = filter p xs, otherwise;

from n = n : from (n + 1);

take 0 xs = [];
take (n + 1) [] = [];
take (n + 1) (x:xs) = x : take n xs;

sieve (x : xs) = x : sieve (filter not_div_x xs)
                 where
                    not_div_x y = (y mod x == 0);
                 ;

primes = sieve (from 2);

main = shownums (take 5 primes);
```

The interaction between the user and the debugger, when the above program is executed, is shown below, with bold typeface indicating the user's input and responses. The questions have been numbered for easy reference. The user may answer "y" for "yes", "n" for "no" and "m" for "maybe" to the questions. He may also issue commands that control various printing options and give assertions; "h" for "help" gives a list of all available options. Note that missing

arguments in applications (i.e. instances of partial applications) are indicated with an underscore ("_") and that an ellipsis ("...") is used to represent tails of long lists and tails that are not evaluated. Also, note that any variables used by a function that are part of its environment, are shown to the user in a `where`-clause (see, for example, question 8). The notation `foo.fie` indicates that function `fie` is declared locally in function `foo`:

```
2 4 8 16 32

1) Did the program behave correctly?
(y/n/m/h)? n

2) main => "2 4 8 16 3..."
(y/n/m/h)? n

3) primes => [2,4,8,16,32,...]
(y/n/m/h)? n

4) (from 2) => [2,3,...]
(y/n/m/h)? y

5) (sieve [2,3,...]) => [2,4,8,16,32,...]
(y/n/m/h)? n

6) (from 3) => [3,...]
(y/n/m/h)? uet
(from 3) => [3:(from ($plus 3 1))]
(y/n/m/h)? y

7) (filter (sieve.not_div_x _ where x=2)
           [3:(from ($plus 3 1))]) =>
[4,6,8,10,12,14,16,18,20,22,...]
(y/n/m/h)? n

8) (sieve.not_div_x 3 where x=2) => False
(y/n/m/h)? n

Bug located in function sieve.not_div_x
```

Clearly, the numbers 2, 4, 8, 16 and 32 are not the first five prime numbers, so the answer to question 1 has to be no. The same holds for questions 2 and 3 as well. The application in question 4 reduces to a list of numbers starting with 2 and 3, which seems to be correct. However, `sieve` applied to this list should evaluate to a list of prime numbers, which is obviously not the case in question 5. The answer to this question is therefore again no.

To be able to answer question 6, the user first requests unevaluated list tails to be shown (the command `uet`). Since `(from 3)` reduces to a list starting with 3 and having `(from 4)` as its tail, this reduction is correct.

The expression in question 7 should filter out everything divisible by 2 from the list of natural numbers starting from 3. Apparently, however, we are left with all even numbers from 4 and upwards, so this is wrong. Finally, in question 8, the offending function is found. Since 3 is not divisible by 2 the expression ought to yield `true`.

For comparison, an unstrictified version of question 2 is given below. It is not too hard to infer the answer to this question, but it is clearly not as straightforward as above. The result of the reduction is a list whose head is the first prime number (2) and whose tail is an expression that according to its specification *should* evaluate to the remaining prime numbers. Therefore, the answer to the unstrictified version of the question is yes, whereas the answer to the strictified question is no since the tail *in fact* evaluates to something erroneous.

```
main =>
['2':
```

```
   ($append
       [' ']
       (shownums
          (take 4 (sieve (filter
                           (sieve.not_div_x _ where x=2)
                           (from ($plus 2 1)))))))))]
(y/n/m/h)? y
```

A larger example may be found in the appendix.

5 Implementation

In this section we present some implementation details on our debugger, LADT. First, in section 5.1, the modifications of the compiler needed to support algorithmic debugging are outlined. Section 5.2 then describes the implementation of the debugger. Section 5.3 explains why perfect strictification is not achieved in the present implementation of the debugger.

5.1 Modifications of the Compiler

As noted in section 4.1, free variables must somehow be dealt with. Being a G-machine implementation, Freja is based on graph reduction using supercombinators[1]. A source program containing functions with free variables is transformed into supercombinator form by lambda-lifting [Jon87][2]. Since only basic lambda-lifting is performed (rather than fully lazy lambda lifting), followed by η-reduction and removal of redundant supercombinators, there is a one-to-one mapping between the functions in the source program and the supercombinators in the object program. This is very convenient, but in a more realistic language this might not be the case, and one would also have to take other source constructs (e.g. list comprehensions) into account.

Therefore, by implementing the algorithmic debugging on the supercombinator form of a program, no extra work is needed to take out free variables; this is already done as a part of the basic compilation process. To support algorithmic debugging, the compiler only has to add some extra debugging information to each supercombinator definition and insert calls to tracing routines in the object code so that an execution tree may be built at runtime. Note that this has to be done for system supplied supercombinators as well, even though we know they are correct, since we need a complete trace tree to perform strictification during the debugging phase.

The extra information that is needed consists of the name of the function corresponding to the supercombinator, the arity of the supercombinator, the number of free variables that have been taken out as extra parameters and the names of those variables. The information is needed so that questions from the algorithmic debugger may be phrased in terms of the original source program rather than in terms of the transformed program, which would be very inconvenient for the user.

Calls to two different tracing routines are inserted in the code of a supercombinator, one call at the beginning and one just before each possible exit point. The routine that is called first creates a node in the trace tree and records the values of all parameters (including the abstracted

1. A supercombinator is basically a function without free variables.
2. Lambda-lifting is the process of abstracting out *free variables* as additional function arguments. In a variant of lambda-lifting, fully lazy lambda-lifting, *maximal free expressions* rather than just free variables are taken out as extra parameters instead.

free variables); the second routine records the result of the supercombinator application. Thus one node in the trace tree will be created whenever a supercombinator reduction is initiated, which happens exactly when the corresponding function is applied to enough arguments, i.e. at least as many arguments as the arity of the function.

In case of a program error (which terminates execution immediately), a call is made to a special routine that first fixes any tree nodes for which no results as yet have been recorded and then immediately invokes the algorithmic debugger. Nodes are fixed by inserting a special result, "error", which is semantically equivalent to a diverging computation, i.e. bottom or $\perp$. If a program goes into an infinite loop, the user may force termination by pressing `Ctrl-C`. The tree is then fixed in the same manner as if a program error had occurred.

The Freja compiler consists of some 10 000 lines of C-code (not counting blank lines and comments). In this respect, the modifications needed for supporting algorithmic debugging were fairly minor, a rough estimate is that about 400 lines had to be modified or added.

5.2 The Algorithmic Debugger

LADT itself is implemented in about 3 000 lines of C. The debugger is linked with the compiled Freja program that is going to be debugged, forming a single executable. When it is run, the Freja program is first executed and the trace tree built. Then the debugger is invoked and algorithmic debugging begins.

The debugger consists of two main parts: routines for building the trace tree, and routines for performing the actual debugging. The tree is constructed by making calls to the two trace routines, as described above. Arguments to supercombinators and returned results are all pieces of graph located on the functional program's own private heap. To preserve them for the execution tree, we chose to copy them off the heap, though other schemes could be devised. The copying algorithm must be able to handle circular graphs; otherwise, the debugger might end up in an infinite loop.

Note, however, that it is not possible to simply keep pointers to the graph pieces, since that would result in substitution of values for expressions without the corresponding changes in the tree structure as explained in section 4.2. But by being a bit cleverer when building the tree, it might still be possible to use this technique, see 7.3.

To facilitate equality testing on graphs, equal graphs are mapped into the same storage using a hashing algorithm, i.e. before any graph node is built, it is checked to see whether an equal graph node already exists. Thus pointer comparison can be used to test for graph equality within the debugger, and the pointers themselves can be used for further hashing when indexing the execution trace tree in order to perform the strictification reasonably efficiently.

5.3 Why Perfect Strictification Is not Achieved

Since the results of all applications that have been evaluated during the program execution have been recorded by the algorithmic debugger, it should theoretically be possible to do a perfect strictification. Unfortunately, this is quite difficult to do using the above approach to strictification.

Consider an unevaluated application `(foo (1+2))` (that would evaluate to `7`, say). Suppose further that the value of `(foo (1+2))` is indeed needed later on, but that the subexpression `(1+2)` is shared with some other expression, the value of which is needed before the value of `(foo (1+2))`is needed. Then the debugger would record that the result of `(foo 3)`

is `7`, but it would know nothing about the result of `(foo (1+2))`. Therefore, it is not possible to substitute `7` for this application during strictification.

In an attempt to somewhat compensate for this imperfectness of the strictification, the debugger explicitly evaluates some system function applications in a few special cases when it is safe to do so. Clearly, better strictification would be preferable.

6 Evaluation

6.1 Experience in Using the Debugger

LADT has so far been used to successfully debug a number of toy programs, including programs calculating Fibonacci numbers and solving the eight queens problem, as well as a somewhat larger program to evaluate arithmetic expressions (around 250 lines of code).

The number of questions that on average have to be answered to find a bug of course varies drastically with the size and type of the program. Finding a bug in the expression evaluator seemed to require some 50–60 questions to be answered on average. There is clearly room for further improvements here.

6.2 Problems with the Current Implementation

The present LADT implementation has two severe problems that make it difficult or impossible to apply to real, large programs. The main one is the size of the trace tree: currently every single reduction is recorded, which means that the size easily could become hundreds of megabytes. The situation is akin to running a lazy functional program without garbage collection. Also, building the tree takes a long time. Though we have not done any extensive measurements, execution seemed to be slowed down by two or three orders of magnitude when doing tracing. This means that it might take a long time before debugging even can start.

Now, it would certainly be possible to keep the trace on secondary storage, even if tracing (as well as strictification) then would become even more expensive. Indeed, one might even argue that the sizes of primary memories within a not too distant future will be large enough to accommodate traces of such sizes. However, the fundamental problem is that there is *no upper bound* on the size of trace, and that problem remains however we store it.

The other big problem is that far too many questions are asked. This makes LADT a very tedious debugging tool to use. This is especially so since it is often obvious to a user that a large number of the questions are irrelevant: it might well be the case that a user can see exactly what is wrong in a result returned from function. If this information could be conveyed to the debugger, rather than just a simple yes or no answer, it should be possible to get to the point quicker, thus discarding many irrelevant questions.

Yet another problem with the current implementation of the debugger is that perfect strictification is not achieved. This means that the user sometimes will have to deal with unevaluated expressions when there should not have been any need to do so. While static strictness analysis probably would make the problem less apparent, strictification is still better than just using static strictness analysis for our purposes.

7 Ideas for Practical Implementation

Despite its problems, we have found the debugger useful, and it should even be possible to debug quite large programs as long as they are applied to test cases of reasonable size. We think that this approach to debugging is basically sound and promising in the context of lazy functional languages. But we would obviously like to do better. Thus, in this section, a number of ideas that address the above mentioned problems and extend the basic approach are outlined, with the aim of making algorithmic debugging for lazy functional languages practical.

The ideas are summarized in the list below and then explained further in the following sections:

- *Thin tracing*: if it is known beforehand that some functions or modules are correct (e.g. library routines), it should not be necessary to trace everything, thereby reducing the size of the trace as well as the number of questions asked.
- *Piecemeal tracing*: do not store the entire trace at once. Instead, parts of the trace can be constructed as and when they are needed by rerunning the program that is being debugged.
- *Immediate Strictification*: instead of building a lazy execution trace tree and applying strictification on it afterwards, a strictified tree could be built directly.
- *Slicing*: if the user is able to be more specific as to what is wrong, rather than just saying that *something* is wrong, it ought to be possible to make use of this information to reduce the number of questions asked by applying program slicing.
- *A smarter user interface*: it is often the case that the details of large data structures are of no interest in answering questions. Thus it would be beneficial to be able to suppress such details.

7.1 Thin Tracing

Presently, LADT records every single reduction that takes place during execution. Quite a few of these are applications of language primitives and library functions which may reasonably be assumed to behave correctly. Thus the user should not be asked about the behaviour of such functions, in which case there seems to be little point in tracing such applications in the first place.

Furthermore, large systems are usually built modularly, so it is not unreasonable to assume that it frequently will be the case that there are a large number of well tested, trusted modules and a few, new “prime suspects” when a bug manifests itself.

So under a *thin tracing* scheme, only a subset of the reductions would be traced, based on assumptions regarding the correctness of certain modules. Clearly, this will also reduce the number of questions that are asked and the time taken to build the tree. However, not having the entire trace at our disposal means that strictification cannot be performed as it is currently done, i.e. during the debugging phase. Therefore, a new approach to strictification must be adopted to use thin tracing.

There is also a more subtle problem as to what is meant by a function being “correct”. For a first-order function it is obvious: a function is correct if it computes the expected result for whatever arguments it is applied to. During debugging, this means that a user no doubt would indicate that an application of the function is correct, and the entire branch in the execution trace tree emerging from that point may thus be cut away.

But for a higher-order function that takes another function as an argument, it would be a rather bold claim that the result is correct for arbitrary arguments: the supplied function is effectively behaviour that has been abstracted out of the higher-order function, and claiming that the higher-order function produces the correct result when some arbitrary behaviour is being plugged back in cannot be justified. It is only possible to say that the higher-order function uses the supplied function in the intended way.

For our purposes, this means that the branch in the execution trace tree corresponding to the application cannot be cut away completely: any applications of the supplied function must be verified.

On the other hand, if it is known that the function that is supplied as an argument is correct as well, then the application could be treated as in the first-order case. This suggests that some simple "correctness calculation" should be performed on higher-order applications.

7.2 Piecemeal Tracing

Even if it is possible to substantially reduce the size of the trace using thin tracing, there is still no guaranteed upper bound on the size of the trace tree. Indeed, as long as the trace for a whole execution is going to be stored, there can be no general, fixed such upper bound.

An interesting alternative would then be to store only so much trace as there is room for. Debugging is then started on this first piece of trace. If this is enough to find the bug, all is well. Otherwise, the program to be debugged is re-executed, and the next piece of the trace is captured and stored. Re-executing the program is not a problem, since pure functional programs are deterministic, but any input to the program must obviously be preserved and reused. We will refer to such a tracing scheme as *piecemeal tracing* from now on.

Note that such a scheme would be beneficial from a time perspective as well: not only does it allow the user to start debugging quicker, but it might also be cheaper overall, since we hopefully avoid tracing of large portions of the execution trace tree[1].

The question is, then, how to select the piece of the trace to be stored in a sensible way? Just storing reductions as they are performed until the trace storage is full would not be very useful since it may then happen that a very deep but narrow piece of the tree is captured. If the top reduction is actually correct, the program would have to be rerun immediately to get the next piece of the trace. It would be better if trace corresponding to the next *n* questions, regardless of what the answers to these questions will be, could be stored.

This leads to the idea of introducing a distance measure on the nodes in the tree based on the number of questions that would have to be answered to get from one node to another (if this is possible, otherwise the measure is undefined). We term this measure the *query distance* and the idea is illustrated in figure 9. The nodes are labelled with their query distance from the root node, and the grey arcs are labelled with the answer that would take a user from one node in the execution trace tree to the other during algorithmic debugging.

Given a cache for execution trace tree nodes, piecemeal tracing can be performed as follows. Suppose that a particular node in the trace tree has been reached during debugging, and that it is found that more trace is needed to proceed. Call this particular node the current node.

Now the entire program is re-executed[2]. Only nodes reachable from the current node (i.e. nodes for which the query distance is defined relative to the current node) are stored in the

1. Presuming that it is possible to make parts of the program for which no trace is stored execute at about their normal speed.
2. There is a good reason for re-executing the *entire* program as will become clear in the next section.

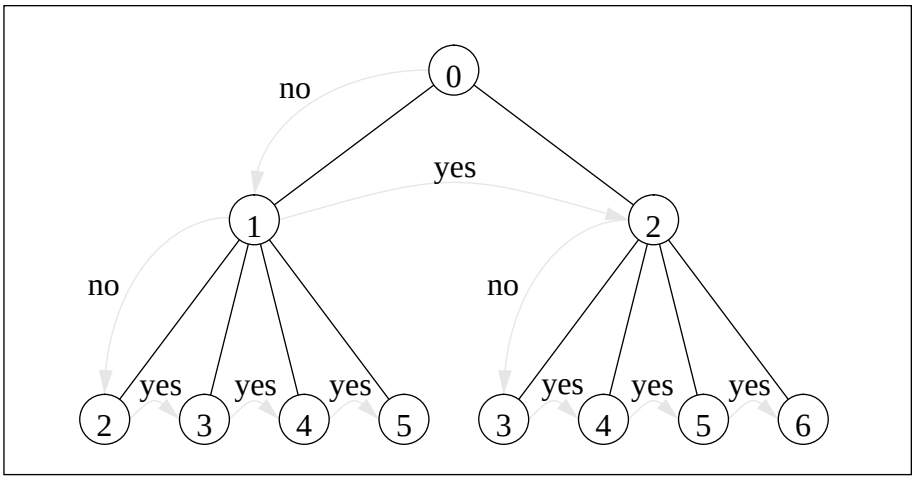

Fig. 9. Query distances from the root node.

cache. When the cache is full, nodes with the largest query distance relative to the current node are thrown out in favour of nodes with a smaller query distance. Then nodes corresponding to the next *n* questions will be in the cache when the tracing is completed, where *n* depends upon the size of the cache.

Thus, an arbitrary upper bound may be imposed on the trace size, but obviously, the larger the size of the cache, the fewer times the program will have to be rerun: the piecemeal tracing scheme makes it possible to trade space for time as is appropriate for any specific implementation. However, as is the case with thin tracing, the entire trace is no longer available during debugging, which means that strictification cannot be performed as it is currently done. This problem is addressed in the next section.

7.3 Immediate Strictification

In the present LADT system, a lazy execution trace tree reflecting the real order in which reductions take place is built during the trace phase. Strictification is performed afterwards, during the debugging phase, by recursively substituting values for expressions while asking for confirmation, as described in section 4.3.

This means that the debugger is effectively redoing work that has already been done during execution of the program that is being debugged. Also, as noted earlier, some strictification opportunities are missed by LADT, which means that only approximative strictification is achieved. Furthermore, doing strictification afterwards is only possible if the entire trace, containing every single reduction, is available to the debugger. Thus thin and piecemeal tracing cannot be integrated with the current implementation of the debugger.

As observed in passing in section 4.2, it would not be necessary to perform any substitutions at all if we kept a pointer from each execution trace tree node to the corresponding piece of graph, since the graph will be in its most evaluated form once the execution is completed. Not only would it then not be necessary to have access to the complete trace during debugging, but it would also solve the problem with approximate strictification. But, as explained in the aforementioned section, correct strictification also requires the structure of the execution tree to be changed. So is it possible to build a tree with the desired, strict structure directly during tracing, rather than afterwards?

Indeed, this seems to be possible. The basic idea is that whenever a *redex* (reducible expression) is *created* during the execution of code corresponding to a function, a node referring to

this redex is also inserted at the appropriate place in the execution trace tree. Since a strict language would not create the redex for later evaluation, but *evaluate it directly*, the "appropriate place" is as a child node of the node corresponding to the current function invocation. The key observation is that the creation of a redex in the lazy case corresponds to strict evaluation of that expression.

The redexes have to be augmented with a pointer referring back to the corresponding node in the execution trace tree so that this node can be found once a redex is reduced since any new redexes created during the reduction should have this node as their parent. Having this back pointer is also convenient during garbage collection, since pointers in the execution trace tree must be updated when pieces of graph are moved around. Note that only a single node in the tree should refer to a particular piece of graph (so that only one back pointer is required): sharing has to be done indirectly via the node corresponding to the function invocation that originally created the redex as illustrated below.

The above also explains why the *entire* program should be re-executed during piecemeal tracing: it is only after completing the execution that each piece of graph will be in its most evaluated form. If we tried to only re-evaluate a particular function application, we would not know when to stop, since we would not know which parts of the result that actually were going to be used later on (if the result is an infinite list, for example).

Hopefully, the following example should give a feeling for what we are trying to achieve. Suppose that we were to execute the following (bug free) program:

```
sqr x = x * x;

main = sqr (1 + 2);
```

Figure 10 shows the corresponding lazy execution tree, i.e. it reflects the order in which the reductions actually happen, while figure 11 shows the strict tree, i.e. the tree that we are trying to construct using immediate strictification.

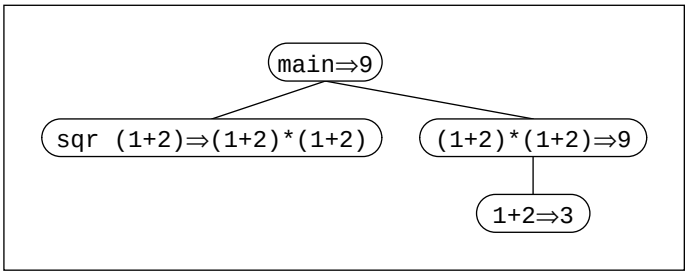

Fig. 10. Lazy execution trace tree

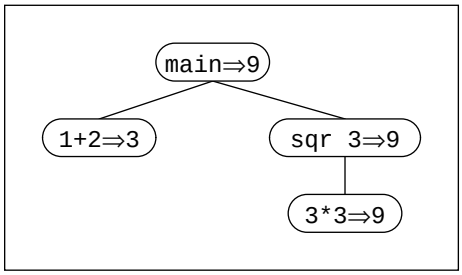

Fig. 11. Strict execution trace tree

First, `main` is invoked. It builds two redexes; `1+2` and `sqr (1+2)`. Thus nodes corresponding to these are inserted into the execution trace tree in the order in which they are created. Note how the `sqr`-node refers to the `1+2`-node so that there is only one pointer from the execution tree to each redex. Since the result of `main` is the result of the latter redex, the result field of the tree node for `main` points at the tree node for `sqr` to preserve the single pointer property. The situation is shown in figure 12.

The next thing that happens, as can be seen from the lazy execution tree, is that the `sqr` function is invoked to reduce `sqr (1+2)` to `(1+2)*(1+2)`. This means that `sqr` has built a new redex, so the corresponding node is inserted into the execution tree. This is depicted in figure 13. Since the result of `sqr (1+2)` is the result of `(1+2)*(1+2)`, the result field of the `sqr` execution tree node has been redirected to the new node.

Then `1+2` is reduced to `3`. This creates no new redexes, but the result of the reduction is copied into the tree (see figure 14). Finally, `3*3` is reduced to `9`. Again, no new redexes are

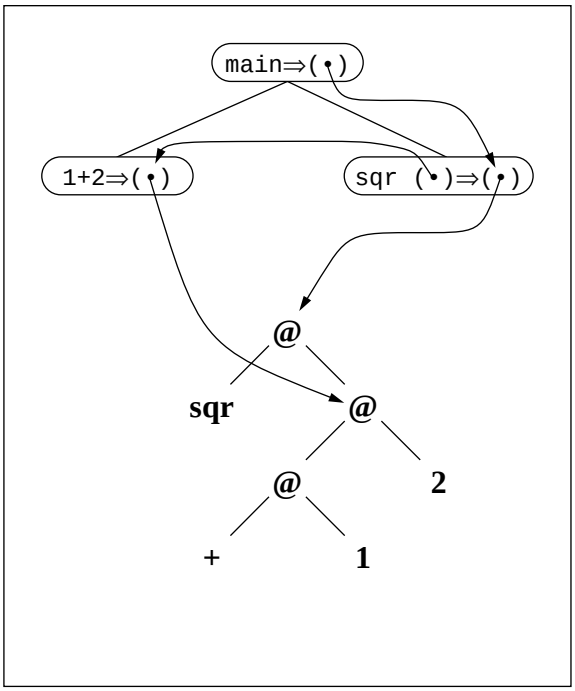

Fig. 12.

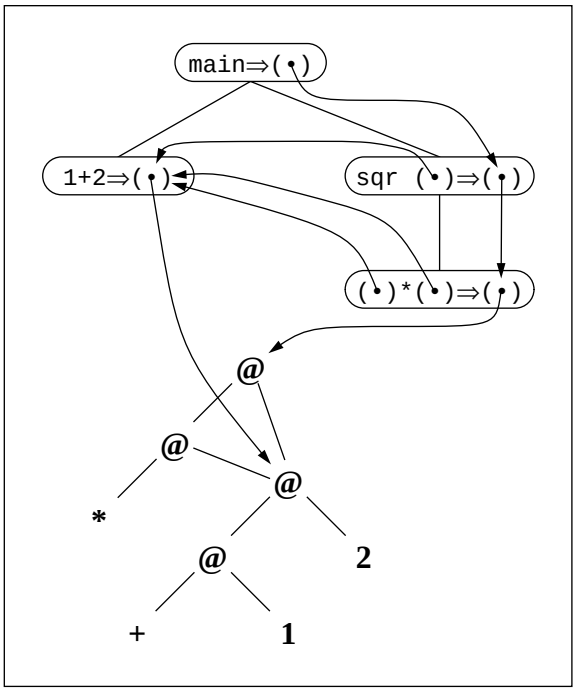

Fig. 13.

created so it only remains to copy the result into the tree (see figure 15). Compare the resulting strictified tree with the strict tree in figure 11.

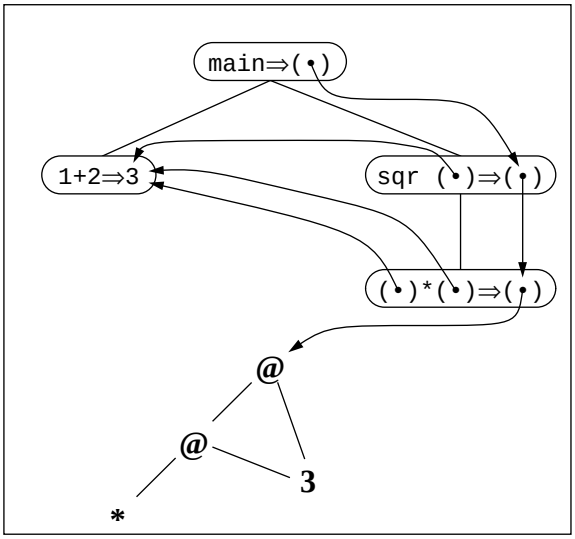

Fig. 14.

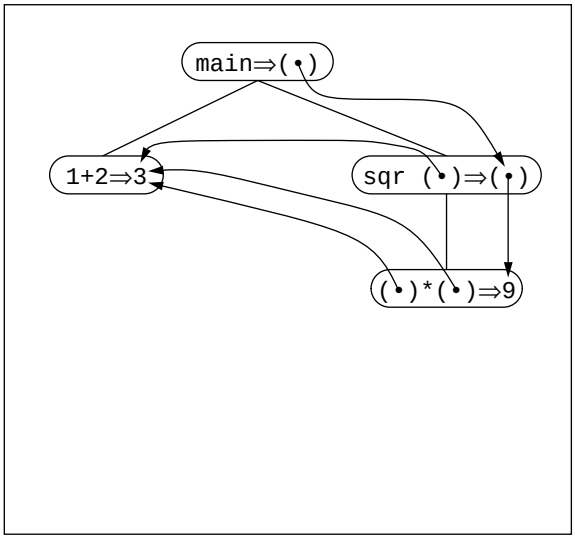

Fig. 15.

7.4 Slicing

Obviously, the more information the user is able to supply the system with per question, the fewer questions the system has to ask to locate the bug. For example, if the user indicates that a particular element in a list is not what it is supposed to be, the system can disregard any computations that are not relevant for the production of this particular element. This technique is a variation of *program slicing* [Wei82][Wei84] and it could reduce the number of questions asked during algorithmic debugging considerably.

The approach that seems to be the most suitable is dynamic slicing, since dynamic slices are more precise than are static ones [Kam93]. Perhaps something similar to what Kamkar has done could be used [KSF92][Kam93]. Amongst other things, this would require keeping track of data dependencies, i.e. which node in the execution tree that corresponds to the function application that computed a certain value. Note, however, that some such information would be present in the strictified tree "for free" if strictification is performed as outlined in the previous section. Thus, it might be possible to do a coarse, approximative, but yet useful, slicing without much extra effort.

7.5 A Smarter User Interface

Even if strictification helps in making the questions asked by the debugger easier to understand and answer, they are still large when large data structures are involved. Frequently, the details of such structures are not important when it comes to answering questions about the correctness of the behaviour of a function application. As a trivial example, consider the function that appends two lists. The actual elements in the list are not important for the behaviour of the function, something which is reflected by the polymorphic type of the function which indicates that it can append lists of any specific type.

This suggests that it might be possible to use type information and other static properties to construct heuristics regarding which parts of large data structures that are interesting, and thus should be presented to the user, and which that are not and thus should be suppressed, at least initially. The gain of a scheme like this could be considerable.

By building a window- and pointer-based graphic user interface, the usability of the debugger would be further enhanced, e.g. suppressed subcomponents could be expanded by clicking on them and erroneous parts of data structures could easily be marked for computing a slice. (See also Westman & Fritzson [WF93].)

8 Related Work

In this section, related work on debugging for lazy functional languages will be reviewed and compared with our work.

Hall & O'Donnell [HO85][OH88] propose a number of approaches to debugging for lazy functional languages in two articles. Their focus is on implementing debugging tools within an interactive, purely functional environment, the main argument for this being portability. They use the language Daisy, a lazy descendant of Lisp.

One suggested approach is to transform the source code of the entire program so that it produces a trace of its execution as well as its normal value. The trace should then be printed before the value of the program so that it is still possible to see something, even if the program happens to loop.

The main problem with this, as Hall & O'Donnell also point out, is of course that the very printing of the trace might turn an otherwise terminating program into a non-terminating one, e.g. if the trace contains references to infinite data structures or to diverging computations, the values of which would not usually be needed. Furthermore, the method presupposes that all types of value, including functional ones, can be printed in a sensible way from within the language. This is not necessarily the case in all functional languages (e.g. Miranda), thus the portability argument is somewhat undermined.

Hall & O'Donnell therefore suggest another approach where the user by means of an interactive debugging function may traverse and evaluate the target program, change variable bindings, evaluate and print expressions in various contexts, etc. Thus the user is allowed to observe, control and modify the execution of a program, all within a purely functional environment. To achieve this, access is needed to the system `eval` function. Also, if the user happens to invoke a non-terminating computation, e.g. by asking for some infinite structure to be printed, it must be possible to interrupt the computation and recover.

Clearly, however, there is still a problem with printing. If the target program is written in a lazy style, chances are that infinite structures and diverging computations will occur frequently, thus making debugging a rather elaborate process involving a lot of trial, error and interrupt. One could conceive special support within the language, e.g. a non-evaluating printing mechanism, but such an *ad hoc* extension would at the best have very dubious semantics.

Toyn & Runciman [TR86][Toy87] argue that even if it is nice to have interactive programming environments, the need for debugging tools for compiling language implementations cannot be neglected. They propose a system in the context of a combinator reduction machine, where the graph is annotated in such a way, and the reduction mechanism modified accordingly, that the computation history of the various pieces of graph on the heap is always maintained as annotations in the graph. Thus it is possible, at any time instant, to take a "snapshot" of the computation, i.e. to present a finite source-level textual representation of the state of the computation. A snapshot would typically be taken when the program aborts due to a runtime error, or when the user interrupts it because it has entered an infinite loop.

It is less obvious how one should get a good, revealing snapshot on a computation that terminates but produces the wrong result. In his thesis, Toyn [Toy87] proposes traditional bottom-up testing to deal with this case.

Kishon *et al.* [KHC91][Kis92] have taken a formal and systematic approach to debugging as well as other monitoring activities, e.g. profiling. Starting from a denotational continuation semantics of an arbitrary language, they automatically derive a monitoring semantics by composing the language specification with a monitor specification. For example, by composing with a tracing specification, the meaning of a program in the language would be changed to be a trace in addition to its original value.

The key difference between this approach and Hall's & O'Donnell's (in the context of lazy functional languages), apart from the former being more systematic and general, is that the computation in this framework is observed from outside the language, which means that there is no problem with printing of infinite structures, etc. On the other hand, unevaluated values will show up in the trace instead, necessitating some kind of work around to get rid of them. Sturrock has tried to use Kishon's framework to build a strictifying algorithmic debugger for a lazy functional language but reports problems [Stu92].

Kamin [Kam90] starts from an operational semantics of a lazy language and changes it so that a program in the language has a tree-structured trace of its execution as its meaning. However, in contrast to Kishon's system, Kamin relies on a meta-evaluation rule to get rid of as many unevaluated values as possible. The rule simply states that values should be shared, i.e. they should be represented by pointers to unique heap-allocated objects. Thus, when the computation has terminated, any value will be seen in its most evaluated form. Since the structure of the trace tree is determined by the syntactic structure of the program (i.e. as would be the case in a strict language) the result resembles very much what we would get under the immediate strictification scheme as discussed in section 7.3.

The general goal of Kamin's work is to demonstrate a hypertextual approach to trace-based debugging. Thus the user is provided with facilities for browsing through the trace, visualize large data structures etc.

Hazan & Morgan [HM93] take a source-level transformational approach to debugging. A program is transformed by their tool so that an explicit call path is constructed and passed around during execution. The path records the static call structure, i.e. it contains information that would be present on a call stack in an implementation of a strict language. However, it contains only the names of the functions, not any parameter values.

The idea is then that the path is appended to any error messages in the code so that in the event of a program error, in addition to the error message, one can see which instance of a function invocation that caused the problem. Note that if the path also contained parameter values, there would again be problems with printing if there happened to be any infinite structures, etc. Though clearly somewhat limited in scope, the tool has been found useful and it is used extensively in the maintenance and development of a large Miranda system (about 12 000 lines of code).

Another debugger that has been used in practice is part of a programming environment for rapid prototyping developed at Technische Hochschule Darmstadt [HKS91]. This debugger, however, is conventional in the sense that it is based on setting breakpoints and single stepping, and it does not address the problem of unevaluated function arguments and results.

Finally, the algorithmic debugger by Naish [Nai92] should also be mentioned. This debugger is for a functional language which is implemented by transformation into Prolog and it is implemented within the prolog system.

It should be noted that several of the above approaches are based on tracing in some form. Kamin even argues that tracing might well be inevitable in the context of lazy functional languages. Thus some of the trace related ideas for making a practical implementation of our algorithmic debugger might also have applications in other types of debuggers. For example, a strictified tree could be used as a basis for more or less ordinary debugging.

9 Conclusions

This paper has argued that algorithmic debugging is a suitable technique for debugging lazy functional programs. An algorithmic debugger, LADT, has been implemented for a lazy functional language, and has been found to be useful in debugging some program examples. A process called *strictification* is performed on the execution trace to give the user an impression of strict evaluation. This makes the debugging process independent of the complexity of lazy evaluation order, and also helps the user to focus on the high-level declarative semantics of the application program.

However, as it stands, LADT cannot be considered to be a practically usable system, mainly due to the prohibitively large trace size for any real world problem, and because far too many questions have to be answered during debugging. Thus we have also suggested a number of techniques in this paper to alleviate these and other problems. *Thin tracing* should be used so that only relevant function applications are traced. By using *piecemeal tracing* the debugger can handle traces of arbitrary size at the expense of re-executing the program when more trace is needed.

Since doing thin and piecemeal tracing mean that strictification cannot be performed during the debugging phase, as is presently the case, a method for building a strictified tree directly during the trace phase, *immediate strictification*, was suggested.

Finally, it was argued that the debugger should employ *dynamic slicing* to reduce the number of questions asked, and that heuristics based on type information and other statically inferable properties should be used to perform sensible pretty printing of questions to reduce their size.

If all these techniques can be successfully integrated, we believe that a practically useful algorithmic debugger for lazy functional languages could be built.

Declaration

This article is based on material that has been presented at PLILP'92 [NF92] and at AADEBUG'93 [NF93]. Basically, these two papers have been integrated and an appendix has been added.

References

[Aug84] Lennart Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 218–227, August 1984.

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languages part II*. PhD thesis, Department of Computer Science, Chalmers University of Technology, December 1987.

[Aug91] Lennart Augustsson. Personal communication on the lack of suitable debugging tools for lazy functional languages, November 1991.

[DNTM88] Wlodek Drabent, Simin Nadjm-Tehrani, and Jan Maluszynski. The use of assertions in algorithmic debugging. In *Proceedings of the FGCS Conference*, pages 573–581, Tokyo, Japan, 1988.

[FGKS91] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmehri. Generalized algorithmic debugging and testing. In *Proceedings of the 1991 ACM SIGPLAN Conference*, pages 317–326, Toronto, Canada, June 1991. A revised version to appear in ACM LOPLAS (Letters of Programming Languages and Systems).

[HKS91] Wolfgang Henhapl, Stefan Kaes, and Gregor Snelting. Utilizing fifth generation technology in software development tools. Report PI-R3/91, Technische Hochschule Darmstadt, March 1991.

[HM93] Jonathan E. Hazan and Richard G. Morgan. The location of errors in functional programs. In *Local Proceedings of AADEBUG'93, 1st International Workshop on Automated and Algorithmic Debugging*, pages 166–182, Linköping, Sweden, May 1993. Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden. Proceedings will also be published by Springer-Verlag.

[HO85] Cordelia V. Hall and John T. O'Donnell. Debugging in a side effect free programming environment. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 60–68, Seattle, Washington, June 1985. Proceedings published in SIGPLAN Notices 20(7).

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the 1984 ACM SIGPLAN Symposium on Compiler Construction*, pages 58–69, June 1984. Proceedings published in SIGPLAN Notices, 19(6).

[Joh87] Thomas Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Department of Computer Science, Chalmers University of Technology, February 1987.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Kam90] Samuel Kamin. A debugging environment for functional programming in centaur. Research report, Institut National de Recherche en Informatique et en Automatique (INRIA), Domaine de Voluceau, Rocquencourt, B.P.105, 78153 Le Chesnay Cedex, France, July 1990.

[Kam93] Mariam Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, April 1993.

[KHC91] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991.

[Kis92] Amir Shai Kishon. *Theory and Art of Semantics Directed Program Execution Monitoring*. PhD thesis, Yale University, May 1992.

[KSF92] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384, Leuven, Belgium, August 1992.

[Nai92] Lee Naish. Declarative debugging of lazy functional programs. Research Report 92/6, Department of Computer Science, University of Melbourne, Australia, 1992.

[NF92] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 385–399, Leuven, Belgium, August 1992.

[NF93] Henrik Nilsson and Peter Fritzson. Lazy algorithmic debugging: Ideas for practical implementation. In *Local Proceedings of AADEBUG'93, 1st International Workshop on Automated and Algorithmic Debugging*, pages 151–165, Linköping, Sweden, May 1993. Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden. Proceedings will also be published by Springer-Verlag.

[Nil91] Henrik Nilsson. Freja: A small non-strict, purely functional language. MSc dissertation, Department of Computer Science and Applied Mathematics, Aston University, Birmingham, England, September 1991.

[OH88] John T. O'Donnell and Cordelia V. Hall. Debugging in applicative languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.

[Sev87] Rudolph E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, 4(3):20–32, May 1987.

[Sha82] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, May 1982.

[Sha91] Nahid Shahmehri. *Generalized Algorithmic Debugging*. PhD thesis, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, 1991.

[Stu92] Robert Sturrock. Debugging systems for lazy functional programming languages. Honours dissertation, Department of Computer Science, University of Melbourne, Australia, November 1992.

[Toy87] Ian Toyn. *Exploratory Environments for Functional Programming*. PhD thesis, Department of Computer Science, University of York, Heslington, York, YO1 5DD, England, April 1987.

[TR86] Ian Toyn and Collin Runciman. Adapting combinator and SECD machines to display snapshots of functional computations. *New Generation Computing*, 4(4):339–363, 1986.

[Tur85] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture, FPCA'85*, number 201 in Lecture Notes in Computer Science, Nancy, 1985.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[WF93] Rickard Westman and Peter Fritzson. Graphical user interfaces for algorithmic debugging. In *Local Proceedings of AADEBUG'93, 1st International Workshop on Automated and Algorithmic Debugging*, pages 300–311, Linköping, Sweden, May 1993. Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden. Proceedings will also be published by Springer-Verlag.

Appendix

In this appendix a larger example is presented. The program below is a simple scanner. The bug is in the function `showtoken`; the case for `Assign` is missing.

```
|| Simple Scanner
||
|| Test program for Algorithmic Debugger
||
|| Ver. 1.00, 1993-05-02

|| ------------------------------------------------------------------
|| Character recognizers
|| ------------------------------------------------------------------
```

```
is_digit x = '0' <= x <= '9';

is_upper x = 'A' <= x <= 'Z';

is_lower x = 'a' <= x <= 'z';

is_alpha x = is_upper x \/ is_lower x;

is_alphanum x = is_alpha x \/ is_digit x;

is_idrtail x = is_alphanum x \/ x == '_';

|| -------------------------------------------------------------------
|| Standard routines
|| -------------------------------------------------------------------

dropwhile p [] = [];
dropwhile p (x:xs) = dropwhile p xs, if (p x)
                   = (x:xs), otherwise;

takewhile p [] = [];
takewhile p (x:xs) = x:(takewhile p xs), if (p x)
                   = [], otherwise;

map f [] = [];
map f (x:xs) = (f x):map f xs;

layn [] = [];
layn (x:xs) = x ++ "\n" ++ layn xs;

foldl op a [] = a;
foldl op a (x:xs) = foldl op (op a x) xs;

|| -------------------------------------------------------------------
|| Conversion routines
|| -------------------------------------------------------------------

char_to_num '0' = 0;
char_to_num '1' = 1;
char_to_num '2' = 2;
char_to_num '3' = 3;
char_to_num '4' = 4;
char_to_num '5' = 5;
char_to_num '6' = 6;
char_to_num '7' = 7;
char_to_num '8' = 8;
char_to_num '9' = 9;

|| -------------------------------------------------------------------
|| Datatypes
|| -------------------------------------------------------------------

type token = Err [char] | Int num | Idr [char] | Plus | Minus | Times |
             Divide | Leftpar | Rightpar | Semicol | Assign | Print | Eof;

is_error (Err msg) = True;
is_error x         = False;

is_eof x = (x == Eof);

|| -------------------------------------------------------------------
|| Show functions
|| -------------------------------------------------------------------

showtoken (Err cs) = "Error: " ++ cs;
showtoken (Int n) = "Int " ++ shownum n;
```

```
showtoken (Idr cs) = "Idr " ++ cs;
showtoken Plus = "Plus";
showtoken Minus = "Minus";
showtoken Times = "Times";
showtoken Divide = "Divide";
showtoken Leftpar = "Leftpar";
showtoken Rightpar = "Rightpar";
showtoken Semicol = "Semicol";
showtoken Print = "Print";
showtoken Eof = "Eof";

show_tokens ts = layn (map showtoken ts);

|| -------------------------------------------------------------------------
|| Scanner
|| -------------------------------------------------------------------------

scan_token::[char]->(token, [char]);

scan_token []           = (Eof, []);
scan_token (' ':xs)     = scan_token xs;
scan_token ('\n':xs)    = scan_token xs;
scan_token ('\t':xs)    = scan_token xs;
scan_token ('|':'|':xs) = scan_token (dropwhile neqlf xs)
                          where neqlf x = (x ~= '\n');;
scan_token ('+':xs)     = (Plus, xs);
scan_token ('-':xs)     = (Minus, xs);
scan_token ('*':xs)     = (Times, xs);
scan_token ('/':xs)     = (Divide, xs);
scan_token ('(':xs)     = (Leftpar, xs);
scan_token (')':xs)     = (Rightpar, xs);
scan_token (';':xs)     = (Semicol, xs);
scan_token (':':'=':xs) = (Assign, xs);
scan_token ('P':'R':'I':'N':'T':xs) = (Print, xs);
scan_token (x:xs) = scan_int (x:xs), if (is_digit x)
                  = scan_idr (x:xs), if (is_alpha x)
                  = (Err ("'" ++ [x] ++ "' not allowed."), xs), otherwise;

scan_int::[char]->(token, [char]);

scan_int xs = (Int val, rest)
        where
          val = foldl mul10add 0 (map char_to_num (takewhile is_digit xs));
           rest = dropwhile is_digit xs;
           mul10add x y = 10 * x + y;
        ;

scan_idr::[char]->(token, [char]);

scan_idr (x:xs) = (Idr (x:idrtail), rest)
        where

           idrtail = takewhile is_idrtail xs;
           rest = dropwhile is_idrtail xs;

        ;

scan::[char]->[token];

scan xs = [tok], if is_eof tok
        = tok : (dropwhile is_error (scan rest)), if (is_error tok)
        = tok : (scan rest), otherwise
```

```
        where
           (tok, rest) = scan_token xs;
        ;

|| -----------------------------------------------------------------
|| Main routine
|| -----------------------------------------------------------------

input = "a := 2; b := 5; a := b * (a + b); PRINT a;";

main = show_tokens (scan input);
```

The interaction between the user and the debugger that follows when the above program is executed is given below. As can be seen, the program only outputs the first token and then gives a runtime error (`PE`, Program Error). The user starts by loading a few assertions and then lists them. Then he starts debugging. The commands `str` and `nostr` switch between printing lists of characters as strings and as lists of characters, while the command `uet` turns on printing of unevaluated list tails.

```
Idr a
PE
Program reduced to Program Error (ie aborted).
Fixing execution tree, will then start algorithmic debugging.

---------------------------------------
Did the program behave correctly?
(y/n/m/h)? load scanner.assert
Assertions loaded from file scanner.assert.
(y/n/m/h)? list
dropwhile
takewhile
map
layn
foldl
char_to_num
(y/n/m/h)? n

---------------------------------------
$main
=>

"Idr a\n..."
(y/n/m/h)? nostr
(y/n/m/h)? uet
(y/n/m/h)? resee
$main
=>

['I','d','r',' ','a','\n':ERROR (execution aborted)]
(y/n/m/h)? n

---------------------------------------
main
=>
['I','d','r',' ','a','\n':ERROR (execution aborted)]
(y/n/m/h)? n

---------------------------------------
input
=>
```

```
['a',' ',':','=',' ','2',';',' ','b',' ',...]
(y/n/m/h)? y
----------------------------------------
(scan ['a',' ',':','=',' ','2',';',' ','b',' ',...])
=>
[(Idr ['a']):(scan [' ',':','=',' ','2',';',' ','b',' ',':',...])]
(y/n/m/h)? str
(y/n/m/h)? resee
(scan "a := 2; b ...")
=>
[(Idr "a"):(scan " := 2; b :...")]
(y/n/m/h)? y
----------------------------------------
(show_tokens [(Idr "a"):(scan " := 2; b :...")])
=>
"Idr a\n..."
(y/n/m/h)? nostr
(y/n/m/h)? resee
(show_tokens [(Idr ['a']):
 (scan [' ',':','=',' ','2',';',' ','b',' ',':',...])])
=>
['I','d','r',' ','a','\n':ERROR (execution aborted)]
(y/n/m/h)? n
----------------------------------------
(is_idrtail ' ')
=>
False
(y/n/m/h)? y
----------------------------------------
(showtoken (Idr ['a']))
=>
['I','d','r',' ','a']
(y/n/m/h)? y
----------------------------------------
(scan [' ',':','=',' ','2',';',' ','b',' ',':',...])
=>
[Assign:(scan [' ','2',';',' ','b',' ',':','=',' ','5',...])]
(y/n/m/h)? y
----------------------------------------
(showtoken Assign)
=>
ERROR (execution aborted)
(y/n/m/h)? n
Bug located in function showtoken.
```