Tracing piece by piece: affordable debugging for lazy functional languages

Henrik Nilsson*

Department of Computer and Information Science, Linkpings universitet, Sweden
and
INRIA Sophia Antipolis, France

Abstract

The advantage of lazy functional languages is that programs may be written declaratively without specifying the exact evaluation order. The ensuing order of evaluation can however be quite involved which makes it difficult to debug such programs using traditional, operational techniques. A solution is to trace the computation in a way which focuses on the declarative aspects and hides irrelevant operational details. The main problem with this approach is the immense cost in time and space of tracing large computations. Dealing with these performance issues is thus the key to practical, general purpose debuggers for lazy functional languages. In this paper we show that computing partial traces on demand by re-executing the traced program is a viable way to overcome these difficulties. This allows any program to be traced using only a fixed amount of extra storage. Since it takes a lot of time to build a complete trace, most of which is wasted since only a fraction of a typical trace is investigated during debugging, partial tracing and repeated re-execution is also attractive from a time perspective. Performance figures are presented to substantiate our claims.

1 Introduction

The distinguishing feature of declarative programming languages, e.g. lazy functional languages such as Haskell, is that they allow the programmer to leave the exact evaluation order in a program unspecified. This is an advantage for the programmer, who can concentrate on the logic of the problem at hand rather than on a detailed description of how to solve it. The result is often very succinct yet clear programs.

Declarative languages also have implications as regards programming errors and debugging. On the one hand, programmers using a declarative language often find that they make fewer programming mistakes than when using conventional languages. This is partly a consequence of the declarative aspects: there are typically fewer lines of code to write and less details to be concerned with, so there are fewer opportunities for making mistakes.

*This work has been supported by the Swedish Board for Industrial and Technical Development (NUTEK) and by the Wenner-Gren Foundations, Stockholm, Sweden.

In Proceedings of the 1999 International Conference on Functional Programming (ICFP), Pages 36–47, Paris, France, September 27–29, 1999.

On the other hand, declarative languages do certainly not eliminate the possibility of making programming errors, so there is still a need to debug code when it turns out that it does not behave as intended. This can be problematic. The reason is that conventional debugging techniques are based on observing execution events as they occur; that is, they are operational. Such techniques are not really useful unless the the programmer fully understands what is going on operationally, which is exactly what a declarative programmer usually prefers not to be too concerned with. It ought to be possible to debug at the conceptual level at which a program is written.

Currently, there are few good debuggers for functional languages [Wad98]. This is true in particular for the lazy functional languages. To this author's knowledge, the only readily available debugging tools which are sufficiently mature for practical use, are either low-level operational tracers (such as the tracing facilities offered by HBC [Aug93]), or specialized tools with a limited scope (for example, Hazan's and Morgan's tool for finding the call path which led to a run-time error [HM93], or Sparud's stream programming debugger [Spa96]). However, there is advanced and ongoing work in this area besides what is presented here, notably the work by Sparud and Runciman [SR97b, Spa99]. See section 5 for further details.

Why are there no good, general purpose, 'lazy debuggers' available? Lazy debugging is difficult for precisely the reasons discussed above: even though the evaluation principle is simple, programmers nevertheless find the ensuing evaluation order 'surprising' [Mor82, OH88]. Because of this, conventional debugging techniques are only of limited use. As a more suitable alternative, a number of researchers have proposed that some form of trace reflecting the *logical* structure of the computation should be constructed, thus allowing debugging to take place at an appropriate level (see e.g. [OH88, Kam90]). However, what were proposed were in most cases *complete* traces. Such traces require storage in proportion to the size of the computation. The result is severe performance problems as soon as realistic programs are being traced.

This paper demonstrates that construction of partial traces on demand, *piecemeal tracing*, can be a viable and efficient way of tracing lazy functional computations. The trace which is constructed is an Evaluation Dependence Tree (EDT) [NS96, NS97] which constitutes a suitable basis for algorithmic debugging [Sha82, NF92]. Since the EDT in many ways resembles a strict call tree, it can also be used to explore a computation in a way similar to how a conven-

tional debugger is used. Furthermore, it is possible to start EDT construction at selected, suspected functions, which is akin to setting break-points in a conventional debugger. The focus of this paper is on performance; the reader is referred to our earlier work [NS96, NS97, Nil98] for detailed descriptions of the EDT and how to build it. Acceptable performance is a prerequisite for a successful debugger based on tracing, and we claim that our performance figures indicate that our scheme is promising in this sense. Whether EDT-based debugging is a good way to debug functional programs is a different matter. We think it is, since it at least shows what is going on inside a program at a suitable level of abstraction, but we do not claim this: ultimately, this can only be decided by user acceptance.

The context of the work is a compiler (called Freja) which currently handles a large subset of Haskell. It compiles to native assembler (SPARC) using a traditional G-machine [Aug84, Joh84] approach. Tracing is achieved by instrumenting the generated code on a supercombinator basis with calls to the tracing routines and code that adds tracing annotations to the graph. The graph annotations take the form of traced application nodes which refer to the point in the EDT where the node corresponding to the reduction of the application should be inserted when and if it is reduced. There are mechanisms for declaring functions and modules to be trusted and to start tracing from suspected functions in order to avoid unnecessary tracing. Language constructs such as list-comprehensions are supported through compilation schemes tailored for debugging. Since our tracing scheme needs a lot of support from the compiler, retrofitting it to an existing system is not trivial. On the other hand, the basic implementation principles are completely standard and used by current Haskell implementations, so integration into other systems is certainly not infeasible.

The structure of the rest of the paper is as follows. Section 2 briefly explains what an EDT is and how an EDT-based debugger works. Section 3 explains how piecemeal tracing works. Section 4 evaluates the performance of the tracer. Section 5 discusses related work. Section 6 sums up and discusses future work. A substantial debugging example is given in the appendix.

2 EDT-based debugging

The Evaluation Dependence Tree is a tree-structured, declarative execution record or trace of a lazy computation. It abstracts from operational details such as evaluation order, emphasising the syntactic structure of the program instead. Each node corresponds to a reduction step, recording the name of the applied function, the arguments and the result. The structure is declarative in the sense that it essentially is a proof tree relating terms to terms through the equations defining the program to be debugged, the *target*. From this perspective, the structure of the tree reflects a proof strategy where terms as soon as possible are simplified exactly as much as needed for obtaining the final result of the program; that is, an eager evaluation strategy which somehow, 'miraculously', stops as soon as the result of a reduction would not be used. Thus one might think of the EDT as a strict call tree, up to a point. This also means that values will be present in their most evaluated form in the tree since reductions seemingly are performed as soon as possible. Furthermore, (sub)expressions left unevaluated can be abstracted to a special value meaning 'unevaluated, assume it is correct' since they cannot have influenced the computation in any way. An important aspect of this tracing strategy is that it does not affect the semantics of the target program: navigation through an EDT does not cause further evaluation.

The following definitions make the notion of an EDT more precise.

Definition 2.1 (Direct evaluation dependence) Let $f\ x_1 \ldots x_m$ be a redex for some function f (of arity m) with arguments x_i, $1 \leq i \leq m$. Suppose

$$f\ x_1 \ldots x_m \ \Rightarrow\ \ldots\ (g\ y_1 \ldots y_n)\ \ldots$$

where $g\ y_1 \ldots y_n$ is an instance of an application occurring in f's body and furthermore a redex for the function g (of arity n) with arguments y_i, $1 \leq i \leq n$. Should the g redex ever become reduced, then the reduction of the f redex is *direct evaluation dependent* on the reduction of the g redex. □

The g redex in definition 2.1 is thus a direct descendant of the f redex (i.e. an instance of an application syntactically occurring in the body of f), and the evaluation of the latter, as far as it was taken, caused the evaluation of the former. Hence direct evaluation dependence. Notice that this is a relation between *reductions*, which also can be seen as function calls. Thus direct evaluation dependence can be understood as a generalized call dependence which does not require the function calls on which a call depends to take place during the latter call.[1] Also note that normal call dependence is subsumed by definition 2.1 as long as it is understood that a direct function call is equivalent to instantiating an application and then reducing it, only much more efficient.

Definition 2.2 (Most evaluated form) The *most evaluated form* of a value is its representation once execution has stopped, assuming a lazy language implementation. □

Definition 2.3 (EDT node) An *EDT node* represents the reduction of a redex. It has the following attributes:

- the name of the applied function
- the names and values of any free variables
- the actual arguments
- the returned result

where values are represented in their most evaluated form. □

Definition 2.4 (EDT) An *Evaluation Dependence Tree* (EDT) is a tree structured execution record abstracting the evaluation order, where:

(i) The tree nodes are EDT nodes (in the sense of definition 2.3), and a special root node which represents the evaluation of the entire program.

(ii) A node p is the parent of a node q if the reduction represented by p is direct evaluation dependent on the reduction represented by q.

[1] Maybe 'lazy call dependence' would have been a more apt description of the relation.

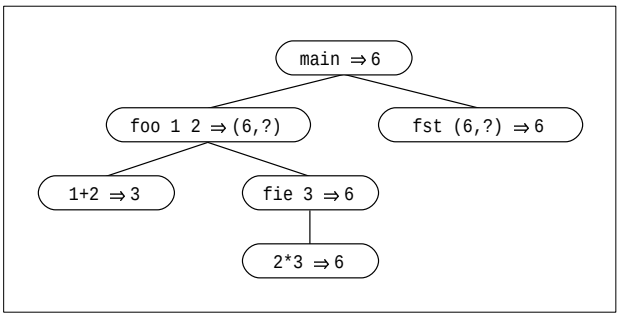


Figure 1: A small EDT. '?' stands for expressions which were never evaluated.

(iii) The special root node is the parent of the EDT nodes representing reductions of top-level redexes.

(iv) The ordering of children is such that a node representing the reduction of an inner redex is to the left of a node representing the reduction of an outer redex w.r.t. the body of the applied function of the parent node. □

The nodes in an EDT may represent only a subset of the reductions which actually were performed in case some functions are trusted. The special root node is needed because there may be many top-level redexes in the form of CAFs besides `main`. Requirement (iv) of definition 2.4 is not a prerequisite for successful debugging, but does ensure that the user gets a chance to verify the computation of arguments before these are used in a call. This is usually helpful.

Figure 1 shows the EDT (excluding the special root node since there is only one top-level redex) for the following Haskell program.

```
foo x y = (fie (x+y), fie (x/0))

fie x = 2*x

main = fst (foo 1 2)
```

Note that the structure of the EDT reflects the structure of the source code, but that only nodes corresponding to reduced redexes are present (e.g. `foo` 'calls' `fie` only once). Also note that some values were never needed, leaving them represented as unevaluated expressions. These are shown as '?'.

Algorithmic debugging is a semi-automatic debugging technique where the debugger tries to locate the node in an execution tree such as an EDT which is ultimately responsible for a visible bug symptom. This is done by checking whether the recorded reductions are correct or not, typically in a top-down order, by asking the user or referring to some formal specification.

As an illustration, suppose the last equation in the above program was `main = snd (foo 1 2)`. This will result in division by zero meaning that the result of the program is undefined, ⊥, which was not intended. The EDT is given in figure 2.

Debugging algorithmically, the debugger would first ask about the result of `main`, which is wrong, then about the application of `foo`, which is wrong as well, then about `1/0` yielding ⊥, which is correct, and finally about `fie` ⊥, also yielding ⊥, which is correct. Given these answers, it would conclude that the bug must be in `foo`.

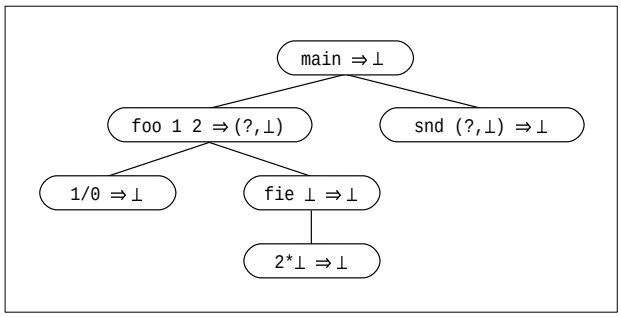


Figure 2: EDT with a bug. ⊥ is the undefined value, in this case `1/0`. An attempt to use it caused an execution error. Since the second component of the result from `foo` was not supposed to be undefined, and since the nodes on which the application of `foo` depends show correct behaviour, the bug must be in the definition of `foo`.

When debugging real code, the questions are often textually large (i.e. the arguments and results are large data structures) and may be difficult to answer. A sophisticated user interface, for example employing configurable graphical visualisation of such data structures, may be a partial remedy. Should the user so desire, for instance because the questions are hard, it is always possible to use a system like this to simply explore the computation, or selected parts of it, in order to gain insight. This is often very helpful in its own right. Given the structure of our EDT, the user would find that this usage mode is not unlike single stepping through normal, imperative code.

A larger example, based on a real debugging session with our debugger, can be found in the appendix.

3 Piecemeal tracing

3.1 Trading time for space

A problem with trace based debugging is that there is no upper limit to the size of the trace. For an EDT-based debugger this is a big practical problem since the logged events (reductions) are very frequent, and since the EDT for efficiency reasons must be held in primary memory. The latter is a consequence of the structure of the computation being different from the structure of the EDT (which is the *raison d'tre* for the EDT). This means that two consecutive reductions can end up in completely different parts of the tree. Thus, in order to carry out the structure transformation simultaneoulsy with the trace construction, as we do, efficient random access is needed to insert a node in the right place in the EDT. Furthermore, arguments and results are not copied from the heap in order to build an EDT node. Instead, for efficiency and to ensure that values are seen in their most evaluated form, the EDT nodes just maintain pointers to arguments and results on the heap. During garbage collection, these pointers must be updated, which again means that efficient random access is needed.[2]

[2] One could imagine a scheme where the structure transformation was deferred by logging execution events to a file as they occur. Individual ETD nodes would then be constructed on demand during debugging. However, this implies that arguments and results would

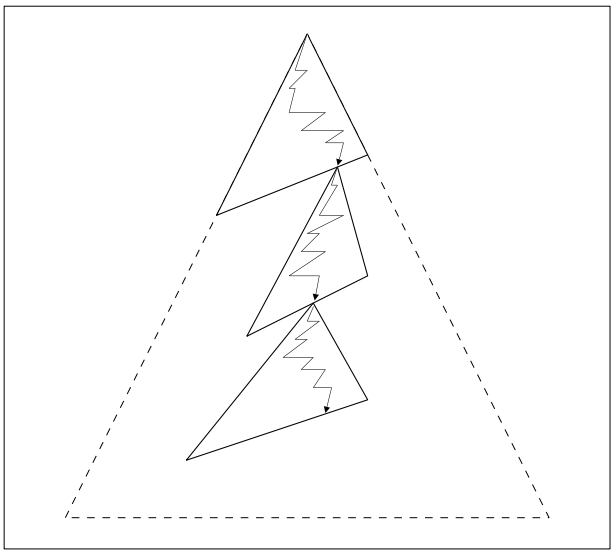


Figure 3: Piecemeal EDT generation. The large, dashed triangle represents the entire EDT, the smaller triangles represent the parts of the EDT which are stored during the first, second and third execution. The path going from the root downwards illustrates how the EDT is traversed during debugging.

In practice, only a small fraction of the execution events are of any interest for finding a bug. Thus it is interesting to use various filtering techniques so as to avoid storing uninteresting events such as reductions involving trusted functions. Indeed, our system has such mechanisms. However, while filtering helps combating the large trace size, and in addition speeds up the debugging process, one cannot expect such techniques to make it possible to carry out arbitrary debugging within limited memory resources: there is no more an upper limit to the number of 'interesting' events than there is one to the total number of events.

An alternative to storing a complete EDT is to store only so much of the EDT as there is room for. Debugging is then started on this first piece of the tree. If this is enough to find the bug, all is well. Otherwise, the target is *automatically* re-executed, and the next piece of the EDT is captured and stored. The user only notices a hopefully not too long delay. We refer to this as *piecemeal EDT generation*. Note that, in the current implementation, re-execution is implemented by running the *entire* program again in order to provide the correct demand context. Re-executing the program is not a problem since pure functional programs are deterministic, even though, from a practical point of view, it is a bit involved since any input to the program must be preserved and reused, a forced termination of a looping program automatically re-issued at the appropriate moment, etc. The process is illustrated in figure 3.

have to be copied to the file, which is very expensive. Based on earlier experience with an implementation along these lines [NF92, NF94], we believe that the postmortem EDT node construction also would be expensive as well as technically complicated. It is interesting to note that for other proposed tracing schemes, notably redex trails [SR97b], it seems to be straightforward to construct the trace on secondary storage but awkward to construct partial traces on demand, i.e. the opposite to what is the case for the EDT.

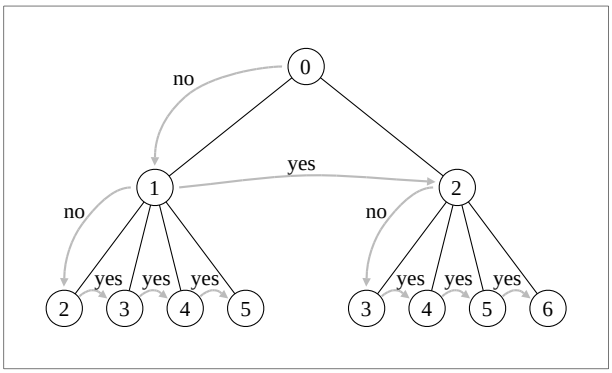


Figure 4: The query distance. The distance between the root and a node is obtained by counting the number of questions that would have to be answered in order to get from the root to the node in question during algorithmic debugging.

3.2 Deciding which nodes to store

For each execution, the EDT nodes which are going to be stored must be selected. We assume that the EDT usually is going to be traversed in an orderly manner, as is the case when debugging algorithmically. This order directly induces a priority order among the nodes: to choose between two nodes, prefer the one which would be visited first. The idea is illustrated in the context of algorithmic debugging in figure 4. Each node is assigned a distance measure relative to the *current* root node by counting the number of questions that would have to be answered in order to get from the current root node to the node in question. We call this measure the *query distance*. Nodes that are close to the root in this sense should thus be preferred over more distant nodes.

Note that answering 'yes' is similar to stepping over a function call in a conventional debugger, and that answering 'no' is similar to stepping into a function call. This is the key to using a debugger like this more like a conventional debugger, and suggests that the query distance is a sensible measure also for this kind of usage.

How many nodes constitute a suitably large EDT piece? The answer is that the number of nodes on its own is not a good measure of the size of the EDT since a single node could refer to an arbitrarily large piece of graph on the heap. A much more robust solution is obtained by monitoring the real memory consumption of the EDT. This allows the size of the EDT to be kept below a certain limit by removing the most distant nodes when the EDT grows too large. In addition to this, there is a (user-definable) upper bound on the number of EDT nodes.

It is important to realize that the size constraint cannot be maintained simply by stopping adding nodes once a size limit has been reached. Instead, the size of the tree must be constantly monitored and the tree must be pruned whenever a limit is exceeded. There are two reasons for this. First, nodes are not inserted into the EDT in an orderly manner, as was explained in section 3.1. This means that the insertion of a node may necessitate the removal of a more distant node to keep the size of the EDT within the prescribed limits. Second, the values referred to from an EDT node may grow *after* the node has been inserted into the EDT. For instance, suppose that we have the following function:

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

Suppose further that there is a redex `from 1`. Once this redex is reduced, the resulting EDT node would refer to the result `1 : from 2`, which is a compact representation of the conceptually infinite list of all integers from 1 and upwards. After a while a much larger part of the result may have been computed, which means that the EDT node refers to a list `1 : 2 : 3 : ...` This representation of the result occupies much more space than the previous one.

The current implementation does not impose an absolutely tight upper bound as the storage requirements intermittently do exceed the prescribed bound: this is what activates the pruning process which keeps the size of the tree within the limits. But as long as a reasonable amount of memory is allocated for debugging (a few megabytes or more), we have found that the scheme works well.

In order to implement a piecemeal scheme as outlined here, we see that the tree construction process must be controlled very carefully and in close co-operation with the runtime system. This kind of control would probably be difficult to obtain if the EDT construction was to be performed at a higher level, e.g. through a transformational scheme (see section 5).

4 Performance evaluation

4.1 Benchmarks and symbols

All measurements have been performed on a 167 MHz Sun UltraSparc 1 equipped with 128 Mbyte of primary memory. Five different benchmark programs have been used. They are all small to average in terms of lines of source code (up to 1000 lines, excluding comments and blanks), but all of them result in substantial computations (the execution times for non-debugged code ranged from 2.4 to 34 seconds and between 1.6 and 16 million *traced* reductions were performed during tracing). Note that the size of the computation rather than the size of the source is what matters here.

- *Sieve*. Computes the 2500th prime number using the sieve of Erathostene.
- *Clausify*. This is a benchmark from the nofib suite [Par93]. Transforms a proposition to an equivalent in clausal form.
- *Cichelli*. From the nofib suite. Uses a brute-force search to construct a perfect hash function for a set of 16 keywords.
- *Parser*. From the nofib suite. Scans and parses 1760 lines of Haskell (the code for the parser itself repeated four times) and prints the resulting abstract syntax tree. This is the largest of the benchmarks in terms of lines of source code (roughly 1000).
- *Mini-Freja*. This is an interpreter for a small functional language. It interprets a program computing a list of the first 500 prime numbers. This is the largest of the benchmarks in terms of the number of traced reductions (16 million).

Table 1 lists and explains the symbols used to denote the various parameters and measured quantities in this chapter. The overall performance is given by relating the total execution time during tracing (t_{tot}) to the corresponding execution time without tracing (t_0). The time for garbage collection makes up a significant part of the execution time during tracing and is thus included in t_{tot}, even though garbage collection times are very sensitive to the amount of memory available and the type of garbage collector used. The other parts of the total execution time are the time spent on performing reductions (t_{red}) and the time spent on constructing the tree (t_{EC}). All of these are accounted for separately in the tables to show where the time is spent and to make it possible to see what would happen if e.g. garbage collection times were significantly reduced.

Symbol	Parameter or measured quantity
T	The number of *traced* reductions. This is a rough measure of the size of a computation. Also note that it is an upper bound of the number of nodes in the *complete* EDT.
N	The number of nodes in the stored part of the EDT at the end of the execution.
$N_{\max}$	User-definable upper bound on the number of nodes in the stored part of the EDT.
RG	Total size of the pieces of graph retained *solely* by the EDT. Note that this is the size towards the *end* of the execution, as measured during the last garbage collection. This does not necessarily reflect the average size of the retained pieces of graph during the execution, or the effort spent on garbage collecting them (i.e. t_{GC}). Moreover, it is not exactly synchronized with N.
$RG_{\max}$	User-definable (soft) upper bound on RG.
t_{tot}	Total execution time; $t_{\mathrm{tot}} = t_{\mathrm{red}} + t_{\mathrm{GC}} + t_{\mathrm{EC}}$.
t_0	Total execution time for the baseline case (no debugging).
t_{HBC}	Total execution time when compiled with HBC to put the baseline case into perspective.
t_{red}	Reduction time. Time spent actually performing graph reduction.
t_{GC}	Garbage collection time. Total time spent on garbage collection.
t_{EC}	EDT construction time. Time spent building and pruning the EDT.
$QD_{\max}$	The maximal QD estimate of a node in the stored part of the EDT. This is an indication of the number of questions that can be answered before the target program is re-executed.

Table 1: Parameters and measured quantities.

Table 2 gives a breakdown of the execution time when the benchmark programs are compiled for ordinary execution with our system. Note that the garbage collection times in most cases are small. This is a result of having a heap which is much larger than the size of the live data. To put the performance of our system into perspective, the column t_{HBC} gives the execution times for the benchmarks when compiled with HBC [Aug97].

4.2 Debugging cost

This section evaluates the performance of our system when performing debugging. The five benchmark programs have been compiled with debugging support and the execution time when building the initial part of the EDT has then been measured for various settings of the parameters $N_{\max}$ and $RG_{\max}$. Table 3 gives the number of traced reductions

Benchmark	t_0 [s]	t_{red} [s]	t_{GC} [s]	t_{GC}/t_0	t_{HBC} [s]
Sieve	7.4	7.3	0.1	0.01	9.6
Clausify	3.5	3.4	0.1	0.03	2.8
Cichelli	9.3	8.7	0.6	0.07	7.9
Parser	2.4	2.2	0.2	0.08	2.1
Mini-Freja	33.5	28.7	4.8	0.14	37.6

Table 2: Breakdown of the execution time for the benchmark programs when compiled for ordinary execution. 10 Mbyte heap, except for Mini-Freja which needed 32 Mbyte. Average times over 5 runs.

for each benchmark.

Benchmark	T
Sieve	6 366 044
Clausify	3 208 175
Cichelli	6 516 387
Parser	1 555 745
Mini-Freja	16 469 854

Table 3: The number of traced reductions for each benchmark.

Note that a program may have to be re-executed several times during a debugging session. Thus, if the debugging cost were to be measured as the total time for all needed re-executions, it would be much larger than indicated here. However, debugging is an interactive activity, so what really matters is response time. The time needed for a single re-execution is therefore more interesting than the total overhead since the former gives an indication of the worst-case response time.

The re-execution frequency should also be taken into account when judging the debugging cost. Provided re-executions do not occur too often, relatively long re-execution times can probably be tolerated since the average response time still would be low. The tables in this section therefore include a column giving the estimated query distance of the nodes furthest from the root of the stored EDT portion ($QD_{\max}$). This gives a rough indication of the number of questions that can be answered before the target program is re-executed.

Of course, if the EDT nodes are not visited in an orderly way, the average response time will increase and approach the worst-case response time. Only extensive real use can determine how much of a problem this is in practice. However, as remarked in section 3.2, the query distance does seem to be a useful metric also when the debugger is used more like a conventional debugger for 'stepping through' an execution. And as long as there are no better alternatives, the inconvenience of a possibly sluggish response has to be weighed against the inconvenience of not having a debugger at all.

Table 4 shows the performance for different values of $RG_{\max}$, the maximal size of the graph retained by the EDT. In each case, the bound on the number of EDT nodes, $N_{\max}$, has been set as high as possible[3] in order that the tree size should be bounded by $RG_{\max}$ only. This is successful in most cases, even if $N_{\max}$ tends to be the limiting factor when $RG_{\max}$ gets large.

Table 4 shows that the time spent on garbage collection increases with increasing $RG_{\max}$. In most cases it quickly becomes the dominating part of the total execution time. The time for building the tree is typically small in comparison, but also tends to grow as the size of the stored part of the tree grows.

Clausify is somewhat problematic since there are very large nodes (nodes which retain a lot of heap, i.e. large arguments or results) close to the root of the EDT. When the debugger tries to keep the size of the tree below $RG_{\max}$, the result is that it throws away almost the entire tree. However, this is not a global property: when starting tracing further down in the tree, it is often possible to store much larger subtrees. The somewhat peculiar entries for N and RG in the first two rows for Parser, are due to N and RG not being exactly synchronized, see table 1.

Table 5 shows the performance for different values of $N_{\max}$, the maximal number of stored EDT nodes. The bound on the size of the retained pieces of graph, $RG_{\max}$, has been set to 64 Mbyte which means that it does not interfere, as can be seen from the column RG.

As the number of nodes in the stored portions of the trees grows, the size of the retained graph grows and so do the garbage collection times. The EDT construction times again grow slowly, but as Sieve, Clausify, and Parser show, EDT construction sometimes account for a significant part of the total execution time.

Table 6 shows the result when $N_{\max}$ and $RG_{\max}$ interact. The bounds have been set to 10 000 nodes and 4 Mbyte respectively. The increase in execution time is below a factor of 3 in all cases, while $QD_{\max}$ indicates that a reasonably large portion of the tree has been stored (except for Clausify).

In conclusion, these benchmarks show that the instrumentation overhead and the cost of building the EDT are low. The costly part of tracing, both in terms of time and space, lies in retaining pieces of graph which otherwise would have been discarded. As the tables show, the time spent on garbage collection can easily account for 75 % or more of the execution time when the retained graph is getting large. This and the fact that memory resources are limited demonstrate the importance of bounding the amount of graph retained by the EDT.

The large overhead for garbage collection is partly due to the use of a simple two-space copying garbage collector. A generational garbage collection scheme would almost certainly be beneficial since it is likely that a large part of the graph retained by the EDT quickly would be moved to an old generation. Earlier experiments carried out in the context of HBC, which has a generational collector, indicate that this indeed is the case [NS96].[4] However, even for a generational collector the garbage collection time increases with the size of the live data.

Furthermore, the results in table 4 and, in particular, table 5 hint at an interesting fact: due to the increasing cost of garbage collection as the size of the stored portion of the EDT grows, it may well be cheaper *overall* to execute a target program a few times with a low bound on the size than

[3] The EDT nodes proper are stored in a table with $N_{\max}$ entries. The memory needed to store this table plus the peak heap consumption must be small enough to avoid excessive paging.

[4] As remarked by one of the anonymous reviewers, it might even be possible to exploit the information about expected lifetime which is inherent in the query distance measure to perform optimizations (distant nodes are likely to die sooner than close ones).

$RG_{\max}$ [Mbyte]	N [nodes]	RG [Mbyte]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\text{tot}}}{t_0}$	$\frac{t_{\text{red}}}{t_0}$	$\frac{t_{\text{GC}}}{t_0}$	$\frac{t_{\text{EC}}}{t_0}$
Sieve								
1	29	0.8	13	16	2.1	1.3	0.4	0.4
2	407	1.6	55	18	2.4	1.3	0.6	0.5
4	6329	3.9	223	29	3.9	1.3	2.0	0.6
8	24978	6.9	446	33	4.5	1.3	2.4	0.8
16	99683	12.4	892	41	5.6	1.3	3.4	0.9
Clausify								
1	18	0.0	12	15	4.3	1.3	1.0	2.0
2	14	0.0	11	17	4.8	1.3	1.3	2.2
4	461	3.3	23	21	6.0	1.3	2.2	2.5
8	329258	5.6	45	23	6.5	1.3	2.2	3.0
16	329258	5.6	45	23	6.5	1.3	2.2	3.0
Cichelli								
1	3079	0.0	58	22	2.3	1.3	0.6	0.4
2	3553	0.0	64	28	3.0	1.3	0.9	0.8
4	4856	0.0	84	50	5.3	1.3	2.8	1.2
8	393695	5.4	153	83	8.9	1.3	6.4	1.2
16	393695	5.4	153	83	8.9	1.3	6.4	1.2
Parser								
1	18366	1.6	123	12	4.9	1.2	2.1	1.6
2	18366	4.9	123	14	5.7	1.2	2.6	1.9
4	74336	6.3	187	17	7.3	1.2	3.6	2.5
8	148839	9.4	265	24	9.8	1.2	5.7	2.9
16	399652	10.2	508	23	9.5	1.2	5.6	2.7
Mini-Freja								
1	31795	0.6	304	74	2.2	1.1	0.8	0.3
2	131862	1.2	609	82	2.4	1.1	1.0	0.3
4	218922	1.8	782	95	2.8	1.1	1.3	0.4
8	799767	5.8	1486	201	6.0	1.1	4.5	0.4
16	799767	5.8	1486	201	6.0	1.1	4.5	0.4

Table 4: Performance for different values of $RG_{\max}$. $N_{\max} = 400\ 000$ except for Sieve where $N_{\max} = 100\ 000$ and Mini-Freja where $N_{\max} = 800\ 000$.

$N_{\max}$ [nodes]	N [nodes]	RG [Mbyte]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\text{tot}}}{t_0}$	$\frac{t_{\text{red}}}{t_0}$	$\frac{t_{\text{GC}}}{t_0}$	$\frac{t_{\text{EC}}}{t_0}$
Sieve								
5000	4952	3.6	198	18	2.4	1.3	0.8	0.3
10000	9872	4.6	280	28	3.7	1.3	2.0	0.4
20000	19902	6.3	398	27	3.6	1.3	1.8	0.5
50000	49772	9.3	630	39	5.3	1.3	3.3	0.7
100000	99683	12.4	892	42	5.6	1.3	3.4	0.9
200000	199398	16.5	1262	52	7.0	1.3	4.5	1.2
500000	499502	21.6	1998	77	10.5	1.3	7.3	1.9
Clausify								
5000	4657	4.7	31	7.4	2.1	1.3	0.5	0.3
10000	7976	4.8	33	7.6	2.2	1.3	0.5	0.4
20000	19565	4.8	36	8.1	2.3	1.3	0.6	0.4
50000	37181	4.9	38	9.8	2.8	1.3	0.8	0.7
100000	97342	5.1	41	12	3.4	1.3	1.1	1.0
200000	182375	5.4	43	16	4.5	1.3	1.8	1.4
500000	431132	5.7	46	25	7.3	1.3	3.5	2.5
Chichelli								
5000	4974	0.0	87	15	1.6	1.3	0.1	0.2
10000	9694	0.5	113	16	1.8	1.3	0.3	0.2
20000	18316	2.1	119	19	2.0	1.3	0.5	0.2
50000	47080	4.3	126	26	2.8	1.3	1.2	0.3
100000	96874	4.5	132	35	3.7	1.3	1.9	0.5
200000	197598	4.8	140	51	5.5	1.3	3.4	0.8
500000	492048	5.6	160	102	10.9	1.3	8.2	1.4
Parser								
5000	4735	3.2	94	4.1	1.7	1.2	0.3	0.2
10000	9962	2.8	108	4.4	1.8	1.2	0.3	0.3
20000	19857	2.0	125	5.1	2.1	1.2	0.4	0.5
50000	49300	4.2	160	7.0	2.9	1.2	0.9	0.8
100000	99223	4.3	215	11	4.4	1.2	1.9	1.3
200000	199537	6.5	338	17	7.1	1.2	4.1	1.8
500000	498262	9.9	567	25	10.6	1.2	6.3	3.1
Mini-Freja								
5000	4930	0.4	126	50	1.5	1.1	0.2	0.2
10000	9952	0.4	174	50	1.5	1.1	0.2	0.2
20000	19978	0.5	243	52	1.6	1.1	0.3	0.2
50000	49841	0.7	378	55	1.6	1.1	0.3	0.2
100000	99710	1.0	531	62	1.8	1.1	0.5	0.2
200000	199565	1.7	747	74	2.2	1.1	0.9	0.2
500000	499217	3.8	1176	124	3.7	1.1	2.3	0.3
1000000	999353	7.2	1660	282	8.4	1.1	6.7	0.6

Table 5: Performance for different values of $N_{\max}$. $RG_{\max} = 64$ Mbyte.

Benchmark	N [nodes]	RG [Mb]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\text{tot}}}{t_0}$	$\frac{t_{\text{red}}}{t_0}$	$\frac{t_{\text{GC}}}{t_0}$	$\frac{t_{\text{EC}}}{t_0}$
Sieve	2487	2.8	140	20	2.8	1.3	1.1	0.4
Clausify	57	0.0	16	7.3	2.1	1.3	0.4	0.4
Cichelli	9694	0.5	113	16	1.8	1.3	0.3	0.2
Parser	9962	2.8	108	4.4	1.8	1.2	0.3	0.3
Mini-Freja	9952	0.4	174	51	1.5	1.1	0.2	0.2

Table 6: Performance for $N_{\max} = 10\ 000$ and $RG_{\max} = 4$ Mbyte.

to execute the same target only once with bounds set sufficiently high to allow the entire tree to be stored. The reason is that only a fraction of the nodes in an EDT typically are visited during debugging, so the re-execution cost is offset by the cost of maintaining irrelevant nodes. If the latter is higher than the former, the piecemeal scheme wins. Had a generational collector been used, the effect might not have been so marked, but it would still be there.

Another interesting fact is that re-execution of the entire target program is not as wasteful as it first may seem: garbage collection and construction of the desired portion of the EDT often constitute the dominating parts of the execution cost. Naish & Barbour [NB95] propose a partial re-execution scheme based on inferring the demand context from the stored result of the application which is re-evaluated. While such a scheme would be beneficial (as long as the gains are not offset by hidden implementation costs), the overhead of garbage collection and tree construction puts an upper bound on the obtainable speedup.

5 Related work

Sparud [Spa96] takes a transformational approach to debugging lazy functional programs. The idea is to transform all functions so that they return an execution record in addition to their normal result. Sparud's aim is to provide a debugging tool which is as portable as possible. However, in order to avoid changing the semantics of the target, a few impure primitives are used. The memory consumption problem is not addressed, and the approach also results in code that runs 8 to 25 times slower than normal not counting the extra garbage collection time [NS96].

The work by Naish and Barbour [NB95] is closely related to Sparud's work [Spa94, Spa96], and there are also similarities to the work presented here. Naish and Barbour use a source-to-source transformation, similar to Sparud's, which transform the target into a program that generates a tree representing a suitable view of the execution in addition to its normal output. A key difference between their transformation and Sparud's is that they rely on an impure function `dirt` (Display Intermediate Reduced Term) which is more complicated to implement than the impure primitives Sparud uses, but which simplifies the transformations. Unfortunately, no performance figures are given.

Naish and Barbour also consider the memory consumption problem and suggest generating parts of the tree on demand. Unlike our piecemeal scheme, they do not require the entire program to be re-executed each time a new part of the tree is needed. Instead, once a node at the fringe of the stored portion of the tree is reached, they re-apply the function of that node to its arguments, and then compare this application to the *evaluated* parts of the result of the previous application of the functions, which is also stored in the node. This will drive the computation exactly the right amount for constructing the tree below the node in question. Note that `dirt` plays a crucial role since comparing against *unevaluated* parts of the result would drive the computation beyond what was originally computed which is unsafe.

As to how much of a tree to store, Naish and Barbour suggest building nodes down to a certain, predetermined, depth. (Then the normal, untransformed, versions of the functions can be called to obtain better performance.) As demonstrated in section 4.2, this does not give a good handle on how much space the stored portion of the tree really occupies, so in general only a few nodes would probably be stored. This in turn could lead to frequent, partial, re-executions, which are not necessarily much cheaper than a complete re-execution.

Recently, Sparud and Runciman have proposed an alternative debugging method based on maintaining complete computational histories for all values [SR97b, Spa99]. They call these histories *redex trails*. The idea is that it should be possible to single out an erroneous value and follow its history backwards until the bug is found. Note that other erroneous values may be encountered during this process, but since *all* values are associated with a trail, it is then just a matter of following one of the other trails instead.

Like Sparud's earlier work [Spa96], the implementation is based on transformations with some support from the compiler. The transformations currently handle most of Haskell. To address the memory consumption problem, Sparud and Runciman propose pruning the redex trails by a modified garbage collector [SR97a]. This risks loosing information important for debugging, but some experiments indicate that this might not be a severe problem in practice. However, the time costs are still too high: executing a traced program takes about 15 times longer than normal. As an alternative to pruning, they also experiment with storing the trace in a file.

6 Conclusions and future work

This paper demonstrated that piecemeal tracing can be an attractive solution to the memory consumption problem for trace-based debuggers for lazy functional languages. A suitable trade-off between time and space can be achieved by setting the size limits for the stored portion of the EDT appropriately. Given a few megabytes of trace storage, a traced program typically takes two to three times longer to execute than normal, while the stored portion of the trace usually is suffiently large to allow reasonably many questions to be asked before it is time to build the next part of the trace. To increase the efficiency further, a generational garbage collection scheme could be used. The current implementation has some problems when large nodes end up close to the root of the stored portion of the EDT. This situation could be detected and the large nodes pruned as a last resort. Other than that, we are currently working on handling full Haskell (problems which has to be solved include handling monadic I/O) as well as methods to speed up debugging in certain cases by providing 'shortcuts' into the EDT.

Appendix: Debugging a small program

In this section, we will demonstrate how our debugger can be used to debug a small but not completely unrealistic lazy functional program. The example is adapted from Johnsson [Joh87], and makes use of a 'circular' programming style which is typical of many lazy programs. Unfortunately, a bug has crept into the adapted code, leading to a black hole.[5]

The purpose of the program which we are going to debug is to take a binary tree where the tips contain elements of a type on which a total order is defined (in our case integers), and return a structurally identical tree where the tips have

[5] The Freja compiler uses circular programming extensively. Black holes, which were quite difficult to track down, were encountered in these parts of the code more than once during the development. If only a debugger had been available!

Productions			Attribute equations		
S	$\rightarrow$	T	$T{\downarrow}itips$	$=$	`[]`
			$T{\downarrow}isorted$	$=$	`sort` $T{\uparrow}stips$
			$S{\uparrow}tree$	$=$	$T{\uparrow}tree$
T	$\rightarrow$	`Tip` x	$T{\uparrow}stips$	$=$	x `:` $T{\downarrow}itips$
			$T{\uparrow}ssorted$	$=$	`tail` $T{\downarrow}isorted$
			$T{\uparrow}tree$	$=$	`Tip` `(head` $T{\downarrow}isorted$`)`
T	$\rightarrow$	T_{L}	$T_{\mathrm{R}}{\downarrow}itips$	$=$	$T{\downarrow}itips$
		`:^:`	$T_{\mathrm{L}}{\downarrow}itips$	$=$	$T_{\mathrm{R}}{\uparrow}stips$
		T_{R}	$T{\uparrow}stips$	$=$	$T_{\mathrm{L}}{\uparrow}stips$
			$T_{\mathrm{L}}{\downarrow}isorted$	$=$	$T{\downarrow}isorted$
			$T_{\mathrm{R}}{\downarrow}isorted$	$=$	$T_{\mathrm{L}}{\uparrow}ssorted$
			$T{\uparrow}ssorted$	$=$	$T_{\mathrm{R}}{\uparrow}ssorted$
			$T{\uparrow}tree$	$=$	$T_{\mathrm{L}}{\uparrow}tree$ `:^:` $T_{\mathrm{R}}{\uparrow}tree$

Figure 5: Attribute grammar for transforming a binary tree into a binary tree with the same shape where the tip values are sorted according to some order. S is the start symbol. $\downarrow$ indicates an inherited attribute, $\uparrow$ a synthesized one.

been sorted according to the total order. However, we wish to do so using only one traversal of the tree using circular programming. The basic idea is that the tree traversal function in addition to the sorted tree returns a list containing the tip values. This list is then sorted and fed back into the tree traversal at the top level. This works fine in a lazy language as long as the traversal is not control dependent on the sorted list.

As Johnsson shows, this is perhaps best understood through an attribute grammar formulation. The grammar can then be transliterated into a lazy functional program, where the laziness ensures proper propagation of inherited and synthesized attributes. An attribute grammar for our problem is given in figure 5. Figure 6 illustrates the attribute propagation for a small tree.

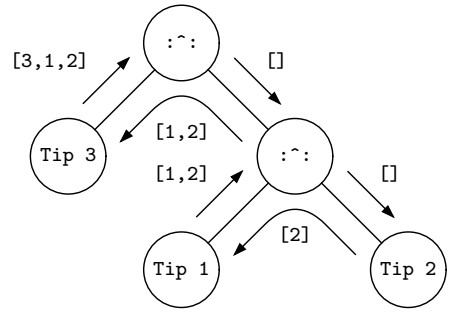
In order to transliterate this grammar into a lazy functional program, one function is introduced for each nonterminal. The functions are defined by pattern-matching over the tree type. There is one case for each of the nonterminal's productions, where the patterns are given by the right-hand sides of the productions in an obvious way. The inherited attributes become additional arguments of the function, and the synthesized attributes are returned as the result, packed into a tuple in case there are two or more. The result of transliterating into Freja is shown in figure 7. The transliteration was performed rather carelessly, however, resulting in a mistake.

When the program is executed, it immediately stops with an error message saying that a black hole has been encountered when evaluating an application of an internal selector function.

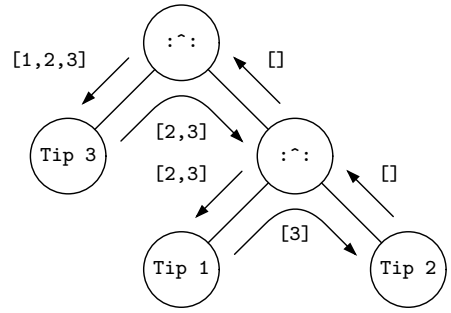
```
[Fatal error] Black hole!
```

Since this does not offer any particularly good lead as to what the problem may be, we recompile the program with debugging support and start it in debug mode. Below, the user's input is typeset in italics.

```
sen1-102% fc -g sorttree.fr
sen1-103% sorttree -- -d
FREJA DEBUGGER
--------------
```



(a) Propagation of *itips* ($\downarrow$) and *stips* ($\uparrow$).



(b) Propagation of *isorted* ($\downarrow$) and *ssorted* ($\uparrow$).

Figure 6: Attribute propagation for a small tree.

```
(Enter "help" to get help.)
[no tree]> debug
(((Tip
[Fatal error] Black hole!
--------------------------------------------
aTree
=>
(:^:)
  ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
  ((:^:) (Tip 3) (Tip 1))
1> yes
```

We use the command `debug` to start a debugging session. The target is then executed, but the execution stops almost immediately with the same error message as before. However, we note that a small part of the result actually has been printed (`(((Tip`). The debugger now proceeds to ask the first question. The question concerns the value of the CAF `aTree`: is it correct or not?. Since the value of `aTree` looks perfectly fine, we answer yes.

```
main => "(((Tip :_|_"
2> no
--------------------------------------------
sortTree
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
=> (:^:) ((:^:) (Tip _|_) ?) ?
3> no
```

```
data Tree a = Tip a | (Tree a) :^: (Tree a) -- deriving Show

sortTree t = t_tree
  where
    (t_stips,t_ssorted,t_tree) = sortTree' t t_itips t_isorted
    t_itips   = []
    t_isorted = sort t_stips

sortTree' (Tip a) t_itips t_isorted =
  (t_stips, t_ssorted, t_tree)
  where
        t_stips   = a : t_itips
        t_ssorted = tail t_isorted
        t_tree    = Tip (head t_isorted)
sortTree' (l :^: r) t_itips t_isorted =
  (t_stips, t_ssorted, t_tree)
  where
    (l_stips,l_ssorted,l_tree) = sortTree' l l_itips l_isorted
    (r_stips,r_ssorted,r_tree) = sortTree' r r_itips r_isorted
    r_itips   = t_itips
    l_itips   = r_stips
    t_stips   = l_stips
    l_isorted = t_ssorted
    r_isorted = l_ssorted
    t_ssorted = r_ssorted
    t_tree    = l_tree :^: r_tree

aTree = ((Tip 7):^:((Tip 2):^:(Tip 5))):^:((Tip 3):^:(Tip 1))

main = print (sortTree aTree)
```

Figure 7: A Freja program for solving the tip sorting problem using only one tree traversal. The program is a transliteration of the attribute grammar of figure 5, but contains a bug.

The next question concerns `main` which evaluated to a string which ends in ⊥. This is not what we expected, so the answer is no. Now the debugger asks about an application of `sortTree`. The argument is OK, but in the result we find ⊥ in a tip. So again the answer is no. We also note that two parts of the result were never evaluated, indicated by the two question marks.

```
sortTree'
  ((:^:)
    ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5)))
    ((:^:) (Tip 3) (Tip 1)))
  []
  ?
=> (?, ?, ((:^:) ((:^:) (Tip _|_) ?) ?))
4> no
```

We are now faced with an application of `sortTree'`. We immediately notice one interesting detail: the third argument (`t_isorted`) was never evaluated (to WHNF). However, this is an operational observation. What does it mean declaratively? Well, we know for sure that an expression which is not evaluated cannot possibly have influenced the computation in any way. In particular, it cannot have caused our black hole. Thus, for the purpose of declarative debugging, we should *assume* that an unevaluated expression represents a correct value!

Continuing with our example, we see, by the same reasoning, that the result as far as we are concerned is mostly correct. However, ⊥ does occur in the result which is not intended. This reduction is therefore incorrect.

```
sortTree'
  ((:^:) (Tip 7) ((:^:) (Tip 2) (Tip 5))) ? ?
=> (?, _|_, ((:^:) (Tip _|_) ?))
5> no
```

The next question is again a call to `sortTree'`. Reasoning as above, we see that the arguments are correct, and we would thus expect the answer to be completely defined. But since ⊥ occurs in the result, this is not the case, and the reduction is again wrong.

```
sortTree' (Tip 7) ? _|_ => ([7:?],_|_,(Tip _|_))
6> yes
```

Once again we encounter a call to `sortTree'`. This time we have to think more carefully about our answer since ⊥ occurs as one of the arguments. The argument in question is `t_isorted`. Looking at the attribute equations for the tip case, we would then expect the returned tip value to be ⊥ (since `head` ⊥ = ⊥) and `t_ssorted` to be ⊥ (since `tail` ⊥ = ⊥). Moreover, we would expect `t_stips` to be a list whose first element is `7`. Thus, given the arguments above, all three components of the result are correct. We therefore conclude that the reduction is correct.

```
sortTree' ((:^:) (Tip 2) (Tip 5)) ? ?
=> (?, _|_, ((:^:) ? ?))
7> no
```

Question 7 is similar to question 5. The answer is again no.

```
sortTree' (Tip 2) ? _|_ => ([2:?],_|_,(Tip ?))
8> yes
---------------------------------------------
sortTree' (Tip 5) ? _|_ => ([5:?],_|_,(Tip ?))
9> yes
```

Questions 8 and question 9 are both similar to question 6, even if the tip values in the results are unevaluated. Both reductions are thus correct.

```
Bug located! Erroneous reduction:
sortTree' ((:^:) (Tip 2) (Tip 5)) ? ?
=> (?, _|_, ((:^:) ? ?))
[no] 7>
```

The debugger has now collected enough information to locate the erroneous function and exhibit a particular application of it which manifests the bug symptom. The bug evidently occurs in the clause for `(:^:)`. Furthermore, we find that the second and third arguments are unevaluated. From an operational point of view, this is strange. Why are these arguments not used? A quick inspection of the source code reveals that `t_isorted` actually does not occur in the body of the second clause of the function. This must be wrong. Looking back at the attribute equations, we spot the mistake and correct the equation for `l_isorted`:

```
l_isorted = t_isorted
```

An alternative approach would have been to inspect the equations in order to find the cause of the black hole, here shown as ⊥. The black hole appears as the second component of the returned tuple, i.e. `t_ssorted` is bound to ⊥. `t_ssorted` is equal to `r_ssorted` which depends on `r_isorted` via the definition of `(sortTree' (Tip 5))`. In turn, `r_isorted` is equal to `l_ssorted` which depends on `l_isorted` via the definition of `(sortTree' (Tip 2))`. But `l_isorted` had by mistake been defined as `t_ssorted`. The definition was thus circular in a self-dependent way, hence the black hole.

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.

[Aug93] Lennart Augustsson. *HBC user's manual*. Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden, 1993. Distributed with the HBC Haskell compiler.

[Aug97] Lennart Augustsson. The HBC compiler. `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`, 1997.

[HM93] Jonathan E. Hazan and Richard G. Morgan. The location of errors in functional programs. In Peter Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 135–152, Linköping, Sweden, May 1993.

[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 ACM SIGPLAN Notices, 19(6).

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173, Portland, Oregon, September 1987. Springer-Verlag.

[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.

[Mor82] J. H. Morris. Real programming in functional languages. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*. Cambridge University Press, 1982.

[NB95] Lee Naish and Tim Barbour. Towards a portable lazy functional declarative debugger. Technical Report 95/27, Department of Computer Science, University of Melbourne, Australia, 1995.

[NF92] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP '92)*, volume 631 of *Lecture Notes in Computer Science*, pages 385–399, Leuven, Belgium, August 1992.

[NF94] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

[Nil98] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden, May 1998. `http://www.ida.liu.se/~henni/thesis.ps`

[NS96] Henrik Nilsson and Jan Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Research Report LiTH-IDA-R-96-23, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden, August 1996. This is an extended version of [NS97].

[NS97] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, April 1997.

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

[Par93] William Partain. The NoFib benchmark suite of Haskell programs. In John Launchbury and Patrick Sansom, editors, *Proc. 1992 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 195–202. Springer-Verlag, 1993.

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

[Spa94] Jan Sparud. An embryo to a debugger for Haskell. Presented at the annual internal workshop "Wintermötet", held by the Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, January 1994.

[Spa96] Jan Sparud. A transformational approach to debugging lazy functional programs. Licentiate thesis, Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden, February 1996.

[Spa99] Jan Sparud. *Tracing and debugging lazy functional computations*. PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden, March 1999.

[SR97a] Jan Sparud and Colin Runciman. Partial redex trails of large functional computations. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL '97)*, St Andrews, Scotland, September 1997.

[SR97b] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *Proceedings of the 9th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '97)*, Southampton, September 1997.

[Wad98] Philip Wadler. An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, February 1998.