How to Look Busy while Being as Lazy as Ever: The Implementation of a Lazy Functional Debugger

HENRIK NILSSON∗
Department of Computer Science, Yale University
(*e-mail:* `nilsson@cs.yale.edu`)

Abstract

This article describes the implementation of a debugger for lazy functional languages like Haskell. The key idea is to construct a declarative trace which hides the operational details of lazy evaluation. However, to avoid excessive memory consumption, the trace is constructed one piece at a time, as needed during a debugging session, by automatic re-execution of the program being debugged. The article gives a fairly detailed account of both the underlying ideas and of our implementation, and also presents performance figures which demonstrate the feasibility of the approach.

1 Introduction

Lazy functional languages have many attractive features. Unfortunately, availability of good, general-purpose debuggers is not one of them (Wadler, 1998). Why is this?

One reason might be that the need for debugging tools for lazy functional languages is less pressing than for traditional languages like C. In any event, it is clear that a lot less effort has been spent on developing debugging[1] tools for lazy languages than, say, to find clever compilation techniques. Hence those who persist in using lazy functional languages have learned to live without the kind of debugging support the rest of the world takes for granted, even further reducing the perceived need for such tools. This is a pity, since the lack of debuggers is likely to be one reason for why lazy functional languages so far have not gained widespread acceptance. Who can blame, say, a student from being discouraged when there is no easy way to find even simple beginners' mistakes? Moreover, even the most experienced and devout functional programmer is likely to waste time looking for bugs which ought to be trivial to find.

∗ This work has been supported by the Swedish Board for Industrial and Technical Development (NUTEK) and by the Wenner-Gren Foundations, Stockholm, Sweden. The implementation work was mainly carried out at Linköpings universitet, Sweden. The paper was mainly written at INRIA Sophia Antipolis, France

[1] Note that we are not concerned with *performance* debugging (or profiling) here. That area has been successfully addressed by others (Runciman & Wakeling, 1993; Runciman & Röjemo, 1996; Sansom & Peyton Jones, 1995).

A more fundamental reason is that it is difficult to develop good debuggers for lazy languages. There are two major problems. The first is inherent in the lazy evaluation strategy. In traditional languages, the execution closely follows the syntactic structure of the source code. This makes it straightforward to trace the computation, set break-points, single step, etc. In a lazy language there is no such immediate relationship between the structure of the source code and the structure of the computation since the computation is demand-driven (Morris, 1982; O'Donnell & Hall, 1988). This is of course why lazy functional languages are interesting in the first place, but it does mean that conventional debugging techniques are only of limited use.

The second problem is the programming style which ensues when the abstraction possibilities offered by functional languages are really exploited, for example in monad-structured code or when using sophisticated combinator libraries. The net effect is typically an abundance of large and complicated functional values, made up from built-in and user-defined functions by means of partial application. Trying to decipher such values during debugging can be quite a challenge. While this problem is not specific to *lazy* functional languages per se, it seems to be the case that this kind of programming idioms are particularly popular within the lazy functional programming community.

However, there is some reason to hope that usable, lazy debuggers could soon be developed. Recent, on-going work in the area includes Sparud and Runciman (Sparud & Runciman, 1997; Sparud, 1999) as well as our own (Nilsson, 1998; Nilsson, 1999). The purpose of this article is to contribute to this end by giving a fairly detailed account of the implementation techniques used in our debugger, and how these mechanisms can be fitted into lazy functional language implementations. While some of the key ideas behind this work are not new (Nilsson & Fritzson, 1994; Nilsson & Sparud, 1997), refining them and putting it all together into a full-scale prototype required a substantial effort and was only completed fairly recently. Thus this is the first time the tested and working versions of our techniques are presented in detail to a broader audience (the only previous detailed description being the author's thesis (Nilsson, 1998)).

The techniques presented in this article address the first of the two problems outlined above, aiming at providing debugging support which is on par with what can be found for other classes of languages. The nature of the second problem is different, and it is a bit difficult to see how a debugger should automatically be able to penetrate arbitrary, user-introduced monads and combinators in order to present a computation at a suitable level of abstraction. Some kind of user extensibility would probably be needed. On the other hand, for a language like Haskell, where all I/O is monadic and where there is syntactic support for programming in monadic style, it is clear that a good debugger at least would have to provide partial solutions in this respect. The implemented system handles higher-order functions and partial application. Thus monadic and combinator-structured code can be debugged, but since there is no specialized support, this is usually not easy. The system does not handle the special monadic syntax in Haskell at all.

Many 'real world' applications tend to rely on commonly implemented language extensions such as unsafe execution of I/O computations, concurrency, and imperative state. In practice, as one of the anonymous referees suggested, it is possible that this is where a large part of any programming mistakes are made. This paper does not address these extensions, but we acknowledge that a comprehensive debugger for a lazy functional language eventually must do so. However, dealing with the purely functional aspects is still going to be central, in particular for users with little previous exposure to functional programming.

Our approach is based on tracing lazy computations abstractly. The trace constructed is an *Evaluation Dependence Tree* (EDT) (Nilsson & Sparud, 1997), which effectively hides most of the artefacts of the lazy evaluation strategy. The EDT, which is a declarative trace, can be used for algorithmic debugging (Shapiro, 1982), or, since it resembles a strict call tree, to explore the computation in a way similar to how a conventional debugger is used. Since construction of a complete EDT in many cases would be infeasible due to time and space constraints, we employ a scheme where parts of the EDT are constructed on demand by re-executing the program being debugged. We refer to this as *piecemeal tracing.* The practicability of this scheme is, of course, to a large part determined by the extent to which it is possible to obtain acceptable performance. We have demonstrated promising results in this respect (Nilsson, 1999). For the sake of completeness, this article also presents some measurements, in some cases for large, realistic, benchmarks which previously could not be tested due to limitations in our compiler.

We do not attempt to evaluate the effectiveness of the debugger in finding bugs in this paper. An independent comparative evaluation of three systems including ours has been carried out by Chitil, Runciman, and Wallace (Chitil *et al.*, 2001). That study indicates that our debugger is fairly straightforward to use, at least for non-monadic and non-combinator-structured code.

The structure of the rest of the article is as follows. Section 2 defines what an EDT is. Section 3 explains the techniques we use to construct them. Section 4 gives a detailed account of our particular implementation of these techniques, and how they have been integrated into a language implementation. Section 5 evaluates the performance of our debugger. Section 6 discusses related work. Section 7 sums up and discusses future work. The appendix explains algorithmic debugging and gives a substantial debugging example.

2 The Evaluation Dependence Tree

The Evaluation Dependence Tree is a tree-structured, declarative execution record or trace of a lazy computation. Its key property is that it abstracts from operational details, such as evaluation order, emphasising the syntactic structure of the program instead. This section defines and illustrates this notion.

2.1 The EDT definition

Each node in an EDT corresponds to a reduction step, recording the name of the applied function, the arguments and the result. The EDT 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 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 are seemingly performed as soon as possible.

(Sub)expressions left unevaluated can, for the purpose of finding some particular bug, be abstracted to a special value meaning 'unevaluated, assume it is correct' since they cannot have influenced the computation in any way. Even if the assumption is wrong, the bug can still be found. This is an important realization, since this abstraction often reduces screen clutter considerably, thereby helping the user focus on the relevant parts of the displayed terms. In the present system, the abstraction is carried out by the user interface, so in principle it would still be possible to display unevaluated expressions if desired.

A crucial aspect of this scheme is that the semantics of the target program is unaffected: construction of and subsequent navigation through an EDT does not cause further evaluation.

The following definitions capture the central aspects of the EDT notion. It might be helpful to read these in conjunction with section 2.2 which provides a concrete example.

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 be 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.[2] 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.

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

Definition 2.2 (Most evaluated form) The *most evaluated form* of a value is its representation once execution has stopped, assuming reduced redexes are updated with the result of the reduction. □

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.
(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. □

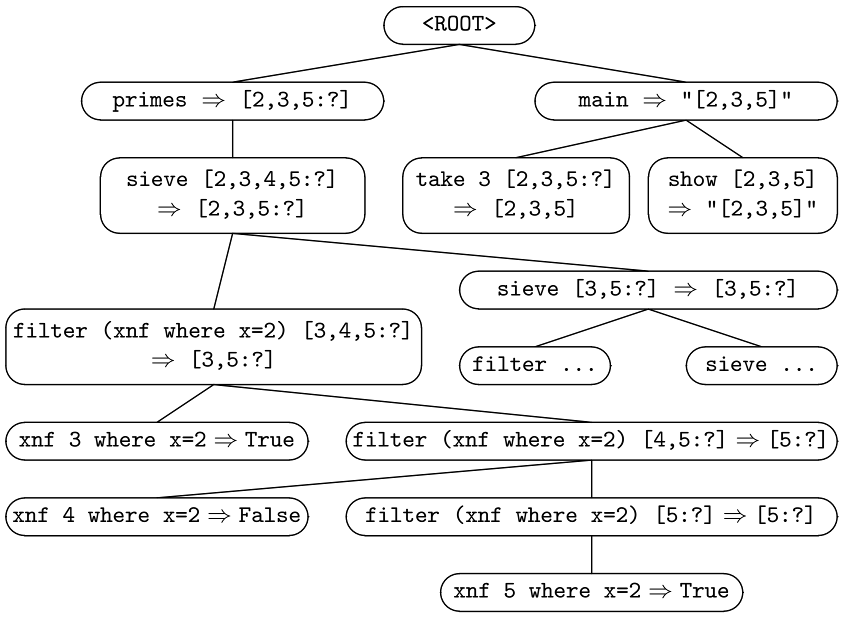
The nodes in an EDT may represent only a subset of the reductions which actually were performed if some functions are trusted. The special root node is needed because there may be many top-level redexes in the form of CAFs (Constant Application Forms) besides `main`. Requirement (iv) of definition 2.4 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. On the other hand, the ordering between the arguments of a function is less important and thus left unspecified.

2.2 The EDT for a small program

As an illustration, figure 1 shows a Haskell implementation of the sieve of Eratosthenes and its EDT. The example illustrates several points regarding the EDT definitions. First, note that the structure of the EDT reflects the syntactic structure of the source code. For example, since `take` is 'syntactically called' from `main`, the `take`-node is one of `main`'s children. For comparison, figure 2 shows what the structure of the actual, lazy, computation might look like. Here, `take` is called by whomever first needs to inspect the result of that redex. However, while the EDT is similar to a strict call tree, there are differences. For instance, `filter` appears to stop calling itself recursively despite not having reached the end of its list argument.

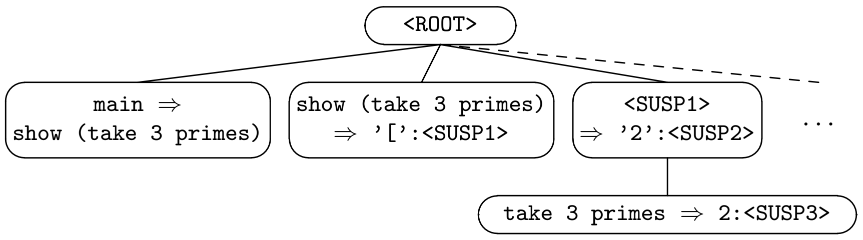
```
sieve (x : xs) = x : sieve (filter xnf xs)
    where
        -- Read "x not a factor"
        xnf y = y `mod` x /= 0
primes = sieve [2..]
main = show (take 3 primes)
```

(a) Sieve of Eratosthenes.



(b) The resulting EDT. '?' stands for expressions which were never evaluated. For space reasons, some parts of the EDT (indicated by ellipses) have been left out.

Fig. 1. Sieve of Eratosthenes and its EDT.



<ROOT>
main ⇒ show (take 3 primes)
show (take 3 primes) ⇒ '[':<SUSP1>
<SUSP1> ⇒ '2':<SUSP2>
...
take 3 primes ⇒ 2:<SUSP3>

Fig. 2. Partial outline of the lazy call tree for the sieve (figure 1(a)).

Second, observe how arguments and results are shown as evaluated as possible. A redex which was never evaluated cannot be shown in normal form since that would imply forcing the redex and thus changing the semantics of the target. Indeed, as illustrated by this example, it is often the case that such redexes do not have any finite normal form. Any remaining redexes are instead indicated by '`?`' meaning that they, for the purpose of debugging, can be assumed to be correct, as discussed above. In contrast, figure 2 shows that what is really returned by, for instance, `main` is not at all the string `"[2,3,5]"`, but a suspended computation which eventually will evaluate to this string. The other suspensions are even more complicated and just shown as `<SUSP`*n*`>`. The interested reader is invited to assume suitable definitions of the involved functions and construct these suspensions in greater detail.

Finally, figure 1 also illustrates how the names and values of free variables are recorded to make it possible to show a closure both when it occurs as an arguments or result (`xnf where x=2`), and when it is applied (e.g. `xnf 3 where x=2`).

The reader might find it helpful at this point to refer to the appendix which contains further examples of EDTs and how they are used.

3 EDT Generation

This section explains the principles of the EDT-generation mechanisms which we have developed. The basic scheme, piecemeal tracing, as well as various optimisations such as selecting starting points and minimising tracing of trusted functions, are discussed. An implementation based on compiled graph reduction is assumed, but apart from this the presentation is fairly general. The choice to carry out tracing at the graph reduction level was made for efficiency reasons and the need to interact closely with the graph reduction process. Alternative approaches are discussed in section 6.

3.1 Dependences

Let us first briefly recall the basic principles of compiled graph reduction. A code sequence is compiled for each function (supercombinator). This code sequence is invoked whenever a saturated application of the function, i.e. a redex, is about to be evaluated. It constructs an instance of the function body and then physically overwrites the redex root with the root of the instantiated body.

For example, suppose that we have a function `f x y = g (h y) x`. The code for `f` will then perform the rewriting step illustrated in figure 3, where `E` and `F` are arbitrary pieces of graph, and the indices on the application nodes (`@`) identifies physical locations on the heap. The redex root is application node 1 in figure 3(a), and it has thus been physically overwritten with the root of the instantiated function body in figure 3(b). The application nodes 3 and 4 are new, constructed by the code of `f` in some newly allocated memory cells, whereas the remaining application node in the original graph, number 2, becomes garbage unless it happens to be shared.

Referring to the EDT definition (definitions 2.1 to 2.4), recall that an EDT node represents the reduction of a redex. It records the name of the applied function,

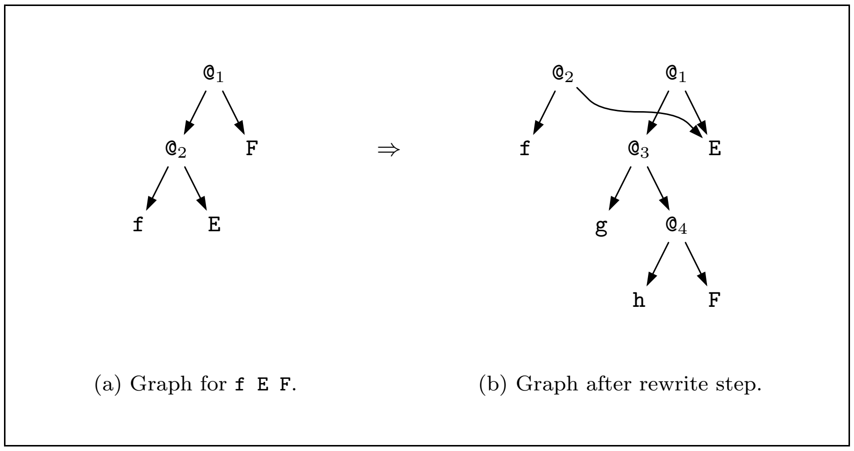


Fig. 3. Graph reduction of `f E F`. The application node indices identify physical heap locations.

the arguments and the result. Furthermore, the evaluation of a redex depends on the evaluation of another if the latter redex is an instance of a function application that syntactically occurs in the body of the applied function.

An EDT node is thus created as the result of *reducing* a redex, not as a result of constructing it. On the other hand, the parent of an EDT node is determined when the corresponding redex is *constructed*. Since the reduction of a redex can occur much later than its construction, it is necessary to keep track of the prospective parent until the redex is reduced (or until the program terminates, at which point it will be clear that the redex will never be reduced).

We solve this by annotating application nodes with a reference to the EDT node in question. Whenever a redex is reduced, the annotation of the application node at the redex root will refer to the node which is the parent of the EDT node which is about to be created. The application nodes are annotated as they are constructed, i.e. during the instantiation of a function body. The annotation refers to the EDT node representing the current reduction, thus capturing the syntactic aspect of the definition of EDT dependence. The EDT node referenced by a redex will sometimes also simply be called 'the parent of the redex', since this EDT node is the record of the reduction that created the redex.

Figure 4 illustrates the above scheme. It is the example from figure 3, but the application nodes have been annotated with references to the EDT (the dashed arrows). As shown in figure 4(a), the redex root of the graph is annotated with a reference to an EDT node marked A. This node is the record of the function call during which the redex `f E F` was built, and it will thus become the parent of the EDT node representing the reduction of this redex.

The situation after the reduction is shown in figure 4(b). A new EDT node, marked B, which is the record of the reduction of `f E F`, has been created and inserted into the EDT as a child of node A. The new application nodes, created as a result of instantiating the body of `f`, have been annotated with references to

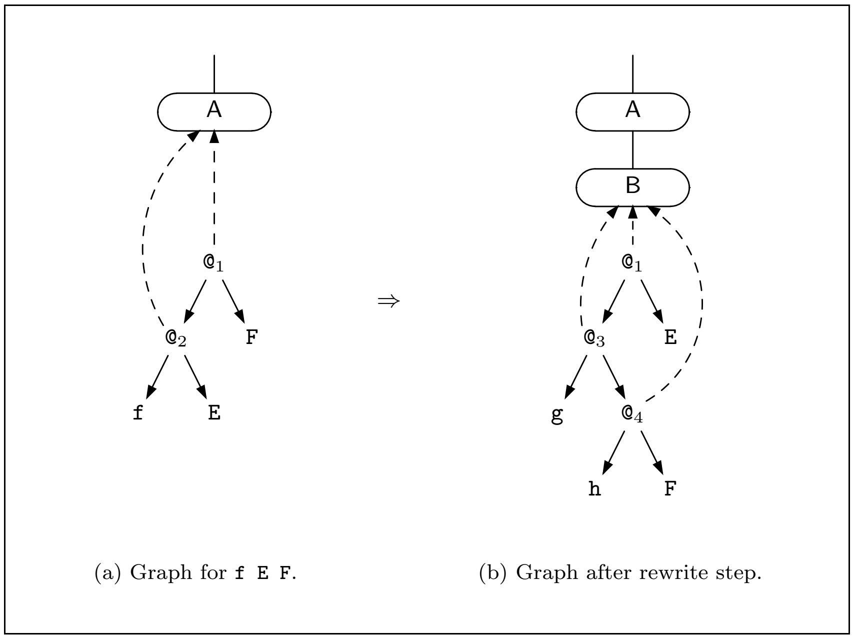


Fig. 4. Graph reduction with annotated application nodes. A and B are EDT nodes. The application node indices identify physical heap locations.

node B. This is also true for application node 1 since it is a new node, even though it is built on top of the old redex root.

This process can be optimised by observing that only application nodes that could be redexes need to be annotated[3]. For instance, if `h` in the above example is a function of arity two or more, then `h F` is not a saturated application. An unsaturated application is not a redex, and there is thus no point in annotating that application node with a reference to the EDT node. Annotated application nodes take more time to construct, occupy more space on the heap, and take longer to garbage collect than unannotated ones.

The children of a node should be ordered innermost-redex first (definition 2.4). But due to the demand-driven evaluation, this is not the order in which nodes chronologically are going to be inserted into the EDT. Application nodes are therefore annotated with a number along with the parent reference as they are created. The numbering reflects the desired syntactical ordering and allows the children of an EDT node to be sorted into order as and when the corresponding redex is reduced.

3.2 Values

The EDT definition also stipulates that function arguments and results should occur in their most evaluated form in the EDT. Consequently, arguments and results

[3] It is not always statically known whether an application is a redex or not.

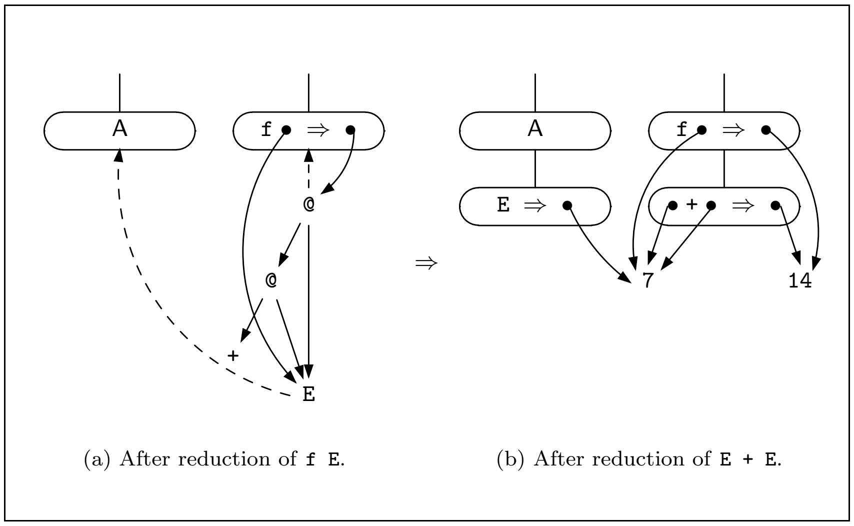


Fig. 5. Handling of values. The arguments and results in the EDT nodes are live pieces of graph, referred to via pointers.

must not be copied from the heap when an EDT node is built, something which in any event would be very expensive, but instead be shared via pointers. Once the execution has terminated, these pointers will refer to the arguments and results in their most evaluated form by the nature of graph reduction.

Keeping references from the EDT[4] to live pieces of graph in the heap means that the garbage collector must be made aware of the EDT. Otherwise values which are part of the EDT but no longer referenced by the target would be lost. As a consequence, not only will the EDT nodes themselves occupy memory, but as new nodes are added, the EDT will also hold on to an increasing amount of heap memory which normally would be reclaimed by the garbage collector. We will return to this point in section 3.4.

The following example shows how arguments and results are handled. Suppose that the function `f` is defined as `f x = x + x`. Suppose also that we are evaluating `f E`, where `E` is a redex and the result of a preceding reduction step. The situation immediately after the reduction of `f E` is shown in figure 5(a). Notice that the argument and the result are live pieces of graph, referred to via pointers from the newly constructed EDT node. Also note how each allocated redex refers to its parent in the tree (the dashed arrows), and that not every application is a redex. The result of `f E` is a new expression, `E + E`, and the next thing that will happen is that this expression is reduced. Now, `+` is strict in both its arguments, which forces the evaluation of `E`, yielding `7`, say.

[4] In the following, the term EDT will be used both to refer to the EDT as previously defined, including the heap-allocated values, and in a narrower sense referring only to the EDT nodes proper. When the distinction is important, it will be clear from the context. This impreciseness is due to the fact that many parts of the graph simultaneously belong to the EDT and to the running program, making the exact extent of the EDT difficult to decide.

The new situation is shown in figure 5(b). The new EDT nodes, representing the reductions of the redexes `E` and `E + E`, have been inserted as children of the redexes' parents. Note how each of these expressions has been overwritten by the value obtained by evaluating it. Thus, once the execution has terminated, values referred from the EDT will be in their most evaluated form.

3.3 Piecemeal tracing

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. 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 simultaneously with the trace construction, as we do, efficient random access is needed to insert a node in the right place in the EDT. Furthermore, the EDT nodes 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.

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 (see section 3.8). 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.

Instead of storing the complete EDT, our solution is to store only so much of it 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. 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 6.

3.4 Deciding which nodes to store

When tracing piecemeal, the EDT nodes which are going to be stored for an execution must be selected. The first step is to select the root of the piece of the EDT which is going to be constructed. This node is called the *current root*, and only

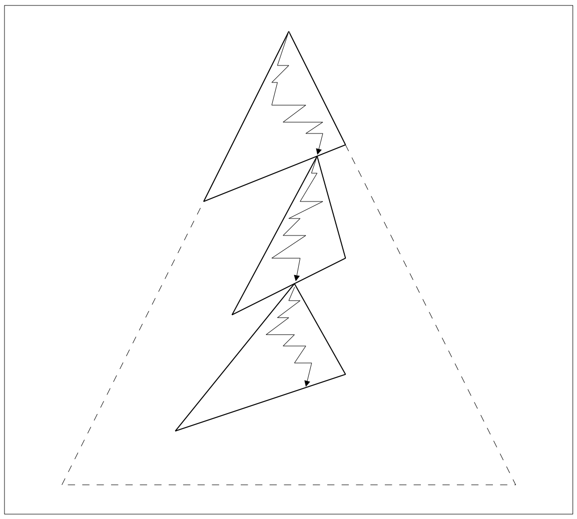


Fig. 6. Piecemeal EDT generation. The large, dashed triangle represents the entire EDT, the smaller triangles represent the parts of the EDT which are stored during the first, second and third execution. The path going from the root downwards illustrates how the EDT is traversed during debugging.

descendants of the current root are eligible for being kept. Then we assume that the EDT usually is going to be traversed in an orderly manner, as is the case when debugging algorithmically or when 'stepping through' a computation. This order directly induces a priority order among the nodes: to choose between two nodes, prefer the one which would be visited first.

Thus the assumption is that during debugging, the next node to be visited is typically either the sibling to the right of the current node, or the first child of the current node. The first choice corresponds to answering 'yes, correct' during algorithmic debugging, or stepping over a function call. The second choice corresponds to answering 'no, incorrect', or stepping into a function call. Under these assumptions, a distance measure relative to the current root can be obtained.

Definition 3.1 (Query distance) The *query distance* measures the distance from the current root to a descendant in terms of the number of questions which have to be answered to get from the former to the latter when debugging algorithmically. Thus:

(i) The query distance to the current root from the current root is 0.
(ii) If the query distance to a node from the current root is d, then the children of that node are assigned increasing query distances, from left to right, starting from $d + 1$.
(iii) The query distance to nodes which are not descendants of the current root is undefined. □

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 heap 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. There is also a (user-definable) upper bound on the number of stored EDT nodes[5]. Notice that an absolutely tight upper bound on the heap consumption is not imposed as the storage requirements intermittently do exceed the prescribed limit: this is what activates the pruning process. But as long as a reasonable amount of heap memory is allocated for debugging (a few megabytes or more), the scheme seems to work well in practice.

It is important to realize that the size constraints 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 (see section 3.3). 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.

3.5 Handling constant application forms

Constant application forms (CAFs) are top-level constants or, equivalently, 0-arity functions. Consistently with the lazy paradigm, the value of a CAF is computed on demand and then shared among its users. The run-time computation of CAFs is what sets them apart from named functions of arity one or more: the latter are always compile-time constants.

From a debugging perspective, the question is where in the EDT the computation of a CAF belongs. A natural solution is to let the computation of each CAF result in a separate EDT since the redexes in the bodies of the CAFs are allocated once and for all at the top-level. However, the CAF computations are not in general independent since a CAF may refer to other CAFs. If the value of a CAF is wrong, the reason may well be that one of the referenced CAFs is erroneous. Unfortunately,

[5] Bounding by the number of nodes occasionally works better than bounding by the size. In addition, the EDT nodes proper are allocated in a pool outside the heap (see section 4.2.3). Knowing the maximal number of nodes simplifies and speeds up the implementation a little.

if each CAF yields a separate, top-level EDT, these dependences are not explicitly recorded and the debugger may conclude that the bug is located in the wrong CAF.

One solution would be to make the CAF dependences explicit by taking out referred CAFs as extra arguments (cf. lambda lifting). However, that would lose CAF sharing. Another solution is to topologically sort the EDTs for the CAFs into dependence order. This ensures that an algorithmic debugger asks the questions bottom-up, as it were, starting with fundamental CAFs which do not depend on other CAFs. An advantage over the first approach is that the questions asked will be simpler. On the other hand, mutually recursive CAFs cause a problem since they cannot be totally ordered while respecting all dependences. Our implementation uses the sorting approach. The problem of mutually dependent CAFs is currently ignored. At the very least, the debugger should warn the user when a CAF belonging to a mutually recursive set is being verified, and in the event that a bug is found in such a CAF, the debugger should list *all* involved CAFs as being potentially erroneous.

3.6 Handling non-terminating programs

Programs are conventionally expected to compute an answer and then terminate. Failure to terminate is regarded as an error. In practice there are programs which are meant to run indefinitely, such as various types of server or control programs, but these are in any event expected to respond to input, or perform some other kind of activity, within bounded time. We will not try to characterise harmful non-termination exactly, but instead assume that it is obvious to a user when a program is non-terminating in the wrong way. In any case, the user is supposed to stop a non-terminating computation, e.g. by pressing some special key combination such as `CTRL-C`, allowing the debugger to gain control.

To ensure transparent re-execution, it is necessary to add a mechanism which automatically stops the target. This can be solved by counting the reductions and stopping when the reduction count reaches the reduction number where the execution was originally stopped by the user.

Semantically, the standard approach is to identify a non-terminating computation with other types of execution errors (division by 0, pattern matching failed, etc.). All of these are denoted by the special symbol $\perp$ meaning undefined. Thus, once a non-terminating program has been stopped, debugging it is in principle no different than debugging any other program. The undefined values are those redexes which were being reduced when the error occurred or the user aborted the execution. Observe that just being a redex is not enough to qualify as being undefined. In an implementation which performs zapping[6] the problem is easily solved since the zapped redexes are exactly those which are being evaluated. Thus a debugger can identify a zap-node with $\perp$.

[6] Overwriting a redex being reduced with a special marker for the purpose of detecting black holes and avoiding a class of space leaks.

3.7 Arbitrary starting-points

During debugging, it is often the case that the programmer focuses on some particular functions, hoping that this quickly will lead to the source of the problem. For instance, these functions might just have been changed, apparently breaking a previously working program. In a traditional debugger, a common method is to set breakpoints at strategic points in the code. Similar functionality is desirable also in an EDT-based debugger. Not only would this save the user the trouble of always having to start debugging from the root of the entire tree, but the number of nodes in the EDT could also be reduced drastically.

Our system allows the user to *dynamically*[7] designate one or more functions as *starting-points*. During tracing, this has the effect of delaying tree construction until an application of one of the starting-point functions is encountered, resulting in a forest of EDTs which are collected below a special root node. Once tracing has begun, functions designated as starting-points are not treated in any special way. In particular, recursive applications of a starting-point function does not result in separate subtrees immediately below the root.

3.8 Trusted functions

Recording every single reduction in the EDT is clearly wasteful. A large number of the reductions are likely to be applications of well-known and trusted built-in functions such as arithmetic operations or functions from the standard libraries (`take`, `drop`, etc.). Moreover, large systems are usually built modularly. So, when a new bug is discovered, a common scenario is that there are a number of well-tested, trusted modules, and a few prime suspects which recently have been added or changed. Hence it would be advantageous if tracing of trusted functions could be minimised.

Shapiro also makes this observation in his seminal paper on algorithmic debugging for Prolog (Shapiro, 1982), and suggests that procedures that have been declared to be 'correct' should not be traced. Shapiro makes use of this to avoid tracing built-in primitives in a sample implementation. This works since Prolog is a first-order language.

Functional languages, on the other hand, are higher-order. Thus it may be statically unknown which function is being applied. This raises the question what it means exactly to 'trust' a function. Does the trust extend to *all* functions which could be applied in a higher-order function like `map`, for instance? If not, and it does seem unreasonable that it would, then it is clear that trusting a function does not necessarily mean that it can be ignored by the tracer since it may be necessary to trace subordinate applications. This is actually an issue in a first-order language

[7] The code generator associates a flag with each supercombinator which is written and read imperatively by the debugger. The desired starting points are given as arguments to the command `debug` which (conceptually) creates the EDT on which to perform debugging. A user can abandon a particular EDT at any point and create a new one by issuing the command `debug` again.

too, since it may not be desirable to extend the trustedness of a function to all its descendants in the (static) call graph.

We take the view that the trust only applies to the actual source code of a function. The meaning of trust thus becomes clear: 'there is no bug in the body of this function.' There is no trust by transitivity: trusting a function does not imply that functions applied in the body become trusted, not even if they are statically known. In the current implementation, the user can declare a module to be trusted through a pragma, indicating that all functions defined in that module are trusted. The compiler then computes an attribute for each function which indicates whether it needs to be traced or not, based on its trustedness and the tracedness of functions applied in its body. The tracedness information is propagated across module boundaries for the purpose of separate compilation. In addition, it is possible to interactively declare further functions to be trusted during debugging. Note that trusting functions with function-valued arguments is no different from trusting first-order functions, except that the former usually end up needing tracing since the tracedness of their function arguments (in general) is unknown.

Now we can make an observation. If untrusted functions are being applied in the context of a trusted function, then the correctness of a call to the trusted function is determined solely by the correctness of any calls to the untrusted functions being applied in its body. In the case of trusted but traced recursive functions, the tracer makes use of this fact and only inserts nodes corresponding to calls of the untrusted functions for recursive invocations. This reduces the the number of nodes in the EDT. See Nilsson (1998) for details.

4 Implementation

This section discusses the implementation of our system. The focus is on the implementation of the EDT generator, but an overview of the entire system is provided first in order to explain the relationships between the different parts and how everything fits together. Our system is based on a traditional G-machine. However, the ideas do carry over to the STG machine as well, and we conclude this section with a brief discussion on this.

4.1 System overview

Our debugger has been developed in the context of a compiler (called Freja) which currently handles a large subset of Haskell[8]. It compiles to native assembler (SPARC) using a traditional G-machine approach (Augustsson, 1984; Johnsson, 1984). The generated object code is linked with the Freja version of the Prelude and the runtime system to form an executable. For debugging, the compiler generates instrumented code, the linking step also includes the debugger, and a traced version of the Prelude is used instead of the standard version; see figure 7. The debugger consists of

[8] Roughly, Haskell less user-defined type classes (standard classes like `Eq`, `Ord` are built in), and less support for monadic programming and monadic I/O.

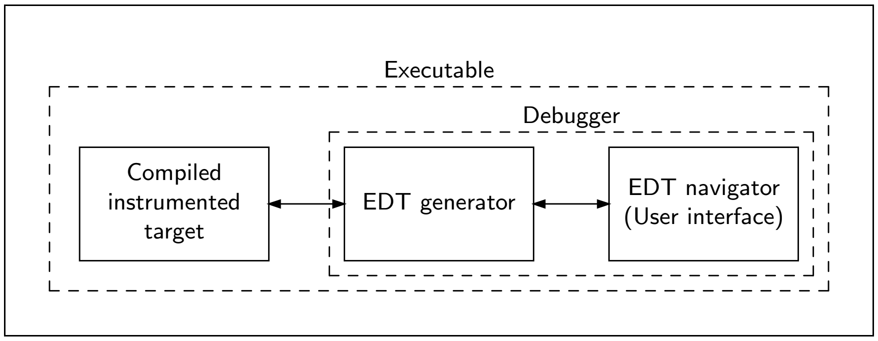


Fig. 7. The components of a debuggable executable. The target program has been compiled for debugging, resulting in instrumented object code. This is linked with the EDT generator and the EDT navigator thus forming a debuggable executable.

the EDT generator and the EDT navigator (a simple user interface). The design of the navigator is such that it easily could be controlled from a separate process implementing a more sophisticated user interface.

4.1.1 The compiler

The compiler itself is written entirely in Haskell (HBC). Scanning and parsing are followed by various transformations and optimisations, type checking, lambda-lifting, generation of G-code, peep-hole optimisation of the G-code, and generation of SPARC assembler. When compiling for debugging, simplified translation strategies are sometimes used and fewer optimisations are performed (e.g. no inlining) to make it easier to relate the generated code to the original source.

The intermediate representations carry debugging attributes which are gradually filled in during the compilation. For example, source code references are maintained (in the form of source code regions) to make it possible for the debugger to show the relevant source code fragments to the user. Other debugging attributes include function and module name, names of any free variables, arity, and attributes related to inference of tracedness (see section 3.8). All attributes are eventually embedded in the generated assembler code in the form of an object information record for each function, making it easy to access the information from the debugger. Some of the attributes are also propagated to the interface files in order to be available during subsequent compilations (e.g. information about tracedness), and to the run-time system generator (see section 4.1.2) through the run-time system dependence files.

Tracing is supported by instrumenting the generated code on a supercombinator basis with calls to the tracing routines in the EDT generator and with code that conditionally, as dictated by the tracing routines, adds tracing annotations to the graph in the form of traced application nodes.

4.1.2 The run-time system generator

The run-time system generator (RSG) generates a large part of the run-time system and computes some debugging-related information according to the needs of the compiled modules. It is run once all modules has been compiled, immediately before linking. Thus the RSG has access to global, cross-module, information. The needs are communicated via special run-time system dependence files which are generated by the compiler. Since it is often the case that many modules have similar needs, the RSG arranges for 'resource' sharing whenever possible. For instance, garbage collection routines are generated and shared extensively: the compiler just tells the RSG what kind of routines a particular module needs (object sizes, CAF dependences). A global dependence analysis is carried out to generate code for performing proper garbage collection of CAFs using a variation of the solution suggested in p. 312.

For debugging purposes, the RSG generates an index which gives the debugger access to all information records for the traced supercombinators. It also computes globally unique function (group) numbers which are used to quickly identify recursive calls during optimised EDT construction as described in section 3.8. In fact, these numbers obtained as a side effect of the global dependence analysis performed to handle garbage collection of CAFs. Section 3.5 explained that the CAF redexes have to be topologically sorted into dependence order. This is also done by the RSG since the dependence analysis has to be carried out across module boundaries, and again the extra implementation effort was minimal since the RSG carries out this analysis anyway as discussed above.

Generally speaking, the RSG has turned out to be a very useful device since it allows a number of tasks to be postponed to 'link time' when information from all modules is available.

4.1.3 The run-time system

The following are the most important points related to debugging:

- The representation of the graph is such that it can be interpreted by the debugger. For example:
  - All objects have distinct tags.
  - Function objects carry the function name[9], the names of any free variables, and source code position.
  - Data objects carry the constructor name and the names of any associated fields.
- The EDT is added to the set of garbage collection roots.
- The garbage collector measures the size of the heap-allocated data which is only referenced by the EDT.
- Control is passed to the debugger on run-time errors and user interrupts.
- Support for re-execution through an interface which allows the debugger to execute and then transparently re-execute the target.

[9] Local functions are given hierarchical names. Lambda-abstractions are given unique, generated, names.

Currently a two-space copying garbage collector is used. It was chosen because of its simplicity.

4.2 The EDT generator

This section describes the implementation of our EDT generator.

4.2.1 Pruning criteria

The strategy we have chosen for pruning was outlined in section 3.4: measure the size of the EDT in terms of its heap memory consumption and the number of EDT nodes; prune the EDT whenever a size limit is exceeded, keeping nodes which are close to the current root according to the query distance. The memory consumption is defined as the amount of heap memory which is only referenced from the EDT; pieces of graph still in use by the running target program do not count. An invariant is that either all or no nodes at a certain distance are present in the stored portion of the tree. This means that once nodes at distance d have been removed, only nodes strictly closer than d are eligible for insertion.

Since it is difficult to know how much memory the pruned tree occupies without performing a new garbage collection, we currently do not check that the pruned tree actually is small enough. However, the tree is pruned substantially at each occasion, by halving the maximum distance of stored nodes, so in a troublesome situation (e.g. when the root node refers to large graphs), the tree will become small quickly. This does mean that the implementation falls short of the ideal of guaranteeing a firm upper bound on the size of the EDT. Furthermore, no attempt is currently made to handle a situation where the root node alone holds on to *more* memory than permitted. The latter actually turns out to be a real problem in some cases, which shows that it may be necessary to prune the arguments and result of an EDT node as a last resort. Of course, that might mean throwing away important information. If so, the user could try to increase the size bound, try to take a shortcut into the tree by using starting points, or try to debug a ‘smaller’ problem, for example.

Computing the query distance exactly according to definition 3.1 turns out to be difficult since nodes are not going to be inserted in an orderly manner from left to right when the tree is constructed. Thus the qd attributes cannot be computed until all nodes have been inserted in the tree, which defeats the purpose of the attribute! This is solved by attributing each *redex* with an estimate of the distance attribute. This estimate can easily be computed given qd of the EDT node for the call which created the redex, and syntactic features of the source code reflecting the innermost-first ordering of the children that we have chosen. When and if a redex is reduced, we take qd of the resulting EDT node to be the estimated distance attribute of the redex. The estimated query distance is also used to sort the resulting EDT node into the correct place among its siblings (see definition 2.4). Using an estimate means that the pruning might be a bit uneven. In the worst case, this results in some extra re-executions.

```
typedef struct edt_node {
   unsigned        id;
   int             qd;
   obj_info        fun_info;
   list            args;      /* list of graph */
   graph           result;
   struct edt_node *parent;
   struct edt_node *leftsib;
   struct edt_node *rightsib; /* Valid if rightsib_qd = 0 */
   int             rightsib_qd;
   struct edt_node *firstchd; /* Valid if firstchd_qd = 0 */
   int             firstchd_qd;
   struct edt_node *next;
   struct edt_node *next_same_qd;
} edt_node;
```

Fig. 8. A somewhat simplified version of the EDT node (C syntax).

4.2.2 EDT node identity

Under a piecemeal generation scheme, only a part of the complete EDT is going to be physically stored in the memory at any one point in time. Yet it must be possible to refer to an arbitrary node in the complete tree. For instance, EDT references from annotated application nodes must be valid regardless of which EDT nodes physically are present.

Each node in the *complete* tree is therefore assigned an identity, *id*, which is independent of which part of the tree is currently stored. One possibility (in a sequential implementation) is to count the reductions. Since the reductions will occur in exactly the same order when the program is re-executed, and since each node in the EDT represents a single reduction, the current reduction count can be used as the *id* of the node corresponding to the latest reduction. A hash table is maintained which makes it possible to quickly find an EDT node given its *id* as long as the node is present.

4.2.3 EDT nodes

Figure 8 shows a simplified version of the EDT node. The EDT generator allocates a pool (an array) of such nodes during initialization, the size of which is decided by the user. The fields `id` and `qd` were described above. The field `fun_info` is a pointer to the information record for the applied function (section 4.1.1). The fields `args` and `result` point to the list of arguments and the result on the heap. The fields `parent` and `leftsib` point to the parent and to the sibling to the left (if any) of the node. They are used to facilitate pruning. The field `next` is used for implementing the hash table which maps an *id* to the corresponding EDT node (if present), and `next_same_qd` is used to link all nodes at the same distance from the current root for pruning purposes.

The remaining fields refer to the sibling to the right (if any) and to the first child (if any). Thus each EDT node refers to a linked list of its children. What

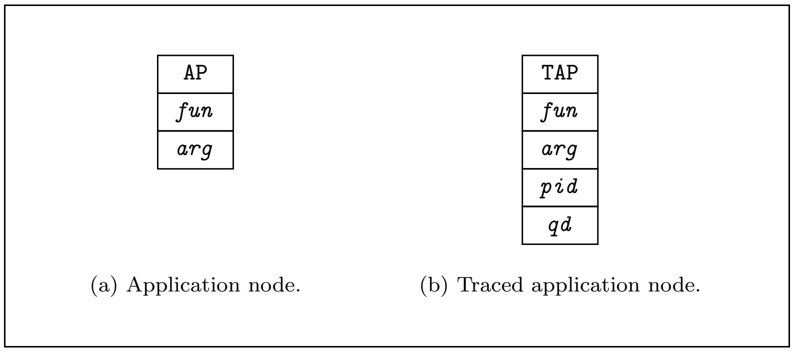


Fig. 9. The layout of untraced and traced application nodes. `AP` and `TAP` are the node tags. The fields *fun* and *arg* are pointers to the applied function and the argument, respectively. The field *pid* is the identity of the parent, and *qd*, finally, is the estimated query distance.

complicates matters slightly, is that it must be possible to refer to nodes which exist in the complete tree but have been pruned away. At the same time, it must be possible to quickly traverse the tree, insert new nodes, etc.

While the *id* of a node, via the hash table, could be used to refer to siblings and children, this was rejected for efficieny reasons. Instead, two fields are used in combination as follows. If `rightsib_qd` is zero, then the field `rightsib` points to the sibling to the right of the node. A `NULL` pointer is used to indicate that there is no sibling. If, on the other hand, `rightsib_qd` is non-zero, then this indicates that the sibling to the right has been removed and that the field `rightsib` is invalid. The value of `rightsib_qd` is the *qd* of the removed sibling (which is strictly larger than zero for all nodes except the root). The fields `firstchd` and `firstchd_qd` work in a similar manner, but refer to the first child of the node.

This might seem a rather indirect way to refer to a missing child or sibling, but the *id* of the *parent* together with the difference between the parent's *qd* and the missing child's or sibling's *qd* make it easy to capture the right part of the tree once an attempt to access a missing node during debugging has triggered a re-execution. The tracer first waits for the the parent *id*, which becomes the new current root, and then for the right child, identified by the *qd* difference. The point of this is that any *siblings* to the right of the desired node will be inserted into the tree (and then possibly removed), making it possible to go both right ('yes' during algorithmic debugging, or stepping over a function call) and down ('no', or stepping into a function call) from the node.

4.2.4 Traced application nodes

A normal application node contains two fields (in addition to a tag): the function being applied and the argument it is applied to. A *traced application node* contains two extra fields: *pid*, the *id* of the *parent*; and *qd*, the estimated query distance. Figure 9 shows the two kinds of application node. A redex where the redex root is a traced application node is called a *traced redex.*

Note that the parent is referred to via its *id* rather than via a pointer. This

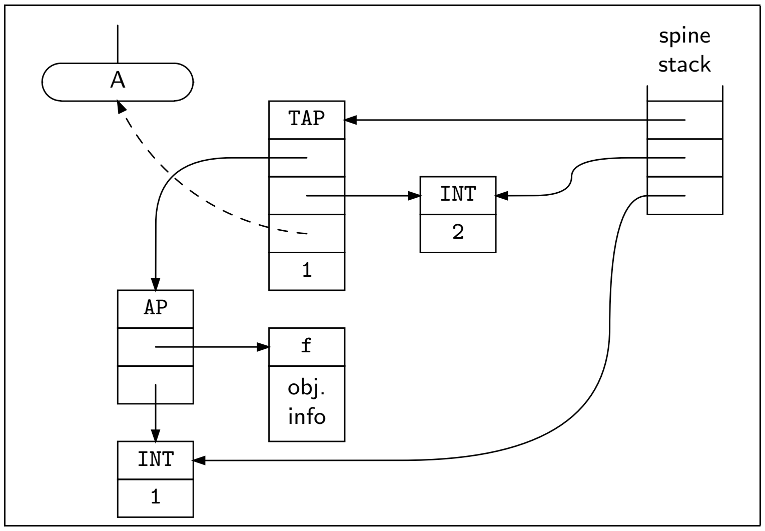


Fig. 10. The reduction of a traced redex `f 1 2`. The figure illustrates the situation just before the code for `f` is entered. It is assumed that the arity of `f` is two. Hence only the redex root is a traced redex.

is because the parent might be removed due to pruning. Had pointers been used, this would result in dangling pointers from the heap into the memory area used for storing the EDT nodes. In general it would then be impossible to determine whether the parent of a redex is present or not. The *id*, on the other hand, is always a valid reference, and the aforementioned hash table which maps *id*s to EDT nodes can be used to find out whether a node is still present.

Figure 10 puts the two types of application nodes into context. It depicts the situation when a traced redex `f 1 2` is being reduced, just before entering the code for `f`. Note the reference from the traced application node (`TAP`) to its parent (the EDT node A), and the estimated query distance which in this case happens to be 1. We are assuming that the function `f` has arity 2. Thus `f 1` is not a redex and the other application node is therefore untraced (`AP`). The box obj. info represents the object information record for `f`. Also note the spine stack which at this stage contains pointers to the redex root and the two arguments.

4.2.5 The Trace algorithm

Now let us consider the algorithm for constructing (a suitable portion of) the EDT. The construction process relies on a subtle interplay between instrumented code, generated by the compiler for each (traced) function, and the EDT construction routines. The scheme has been designed in such a way that instrumented and uninstrumented code (i.e. code for traced functions and untraced functions, respectively) is interoperable. Entering uninstrumented code disables EDT construction below that point. This is exactly what we want for trusted, first-order functions not calling functions which need tracing. On the other hand, whenever EDT construction

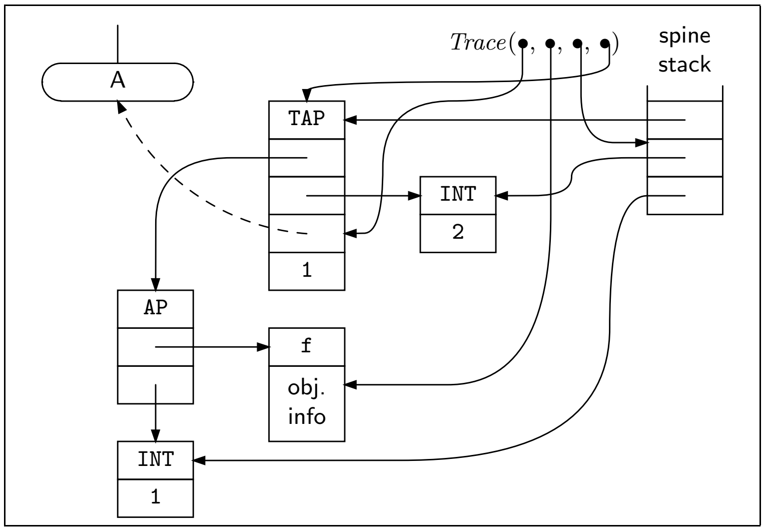


Fig. 11. The reduction of a traced redex `f 1 2`. The figure illustrates the situation just before *Trace* is called from the code for `f`.

is desired, the involved function *must* be traced (even if trusted; see section 3.8). The complete EDT only contains nodes for calls to traced functions.

The following main steps are performed when a traced function is invoked:

1. Call *Trace*, the main EDT construction routine, with the following arguments:
   - A pointer to a record containing the *pid* and *qd* from the redex root. `NULL` if the redex root is untraced.
   - A pointer to the object information record.
   - A pointer to the argument vector on the spine stack.
   - A pointer to where the result will be located, i.e. the redex root.

   Figure 11 illustrates how *Trace* is invoked. It shows the reduction of a traced redex `f 1 2`, just before calling *Trace*. Compare figure 10.
2. *Trace* now determines whether or not to build an EDT node (see below). If a node is built, the *id* of that node is returned along with an initial estimate of *qd* for constructing traced redexes. Otherwise a special value *nt* ('no tree') is returned. This signals that no traced redexes are to be constructed, causing the code to behave exactly as if it had not been instrumented. The effect of this is to disable EDT construction below this point (unless *Trace* overrides, see below).
3. Whenever a (potential) redex is built and tree construction is enabled, it is made a traced redex where *id* is set to the *id* returned from *Trace*, and *qd* is set to the initial *qd* plus an offset. This offset reflects the desired innermost-first ordering of child redexes and could be computed statically at compile time. However, in our case it was easier to use a counter, initialized to the initial *qd* estimate, which is incremented by one each time a traced redex is built. This works since redexes are constructed in a suitable order.

Trace has two modes. It starts in waiting mode where it stays until the reduction which corresponds to the root of the desired EDT part (the current root) takes place. Then it enters construction mode in which the actual tree building is performed. In waiting mode, *Trace* performs the following steps:

1. Increment the reduction counter.
2. Decide whether it is time to enter construction mode. This is just a matter of comparing the reduction count with the *id* of the current root. If it is not yet time, return *nt* to the caller.
3. Create the root node. Its *id* is set to the current value of the reduction counter, which is also returned to the caller along with the initial *qd*.

Note that instrumented functions always call *Trace*. Thus *Trace* can initiate tree construction whenever that is appropriate. Traced redexes are *not* constructed from the top down to the current root (which would be expensive).

In construction mode, *Trace* behaves as follows:

1. Increment the reduction counter.
2. Check whether the redex root is a traced application node. If not, tree construction is disabled below this point and *nt* is returned to the caller.
3. Get *pid* from the redex root and check if node *pid* is still present. If not, return *nt* to caller.
4. Get *qd* from the redex root and check if it is small enough. If not, return *nt* to the caller.
5. Prune the tree if necessary in order to insert a new node. This must be done since the EDT nodes are allocated from a pool whose size is determined at the start of a debugging session. Thus it may be the case that there are no free EDT nodes, and if the node which is about to be inserted is closer to the current root than the most distant node in the tree, then the tree should be pruned to make room for the new node. This should not be confused with the pruning performed by the garbage collector.
6. Create and insert a new EDT node among the children of node *pid* at the correct place as indicated by *qd* (children are sorted by *qd*). The field `id` is set to the current value of the reduction counter, which is also returned as *id* to the caller along with *qd* from the redex root. The fields `fun_info`, `args`, and `result` are set from the corresponding actual arguments supplied to *Trace*.

For further details, including the source code for *Trace* with explanations, see Nilsson (1998).

4.2.6 Integration into the G-machine

In order to accommodate the tracing machinery within the G-machine framework (Augustsson, 1984; Johnsson, 1984; Peyton Jones, 1987), two small modifications of the machine were made. First, the instruction `FUNSTART` was changed so that *Trace* is called for traced functions as explained earlier. It also allocates two variables (on the dump) where it stores the returned *id* and the initial *qd*. Second, a new instruction `MKTAP` was introduced to build traced application nodes. It behaves as follows:

1. Check *id* (stored on the dump by `FUNSTART`). If it is the special value *nt* ('no tree'), build a normal application node (i.e. behave as the instruction `MKAP`).
2. Otherwise, build a traced application node. The *pid* field is set to *id* (the EDT node *id* is the parent of the redex), and the *qd* field is set to the current value of the variable *qd* (on the dump).
3. Increment the variable *qd* by one. This ensures that traced redexes are assigned increasing estimated query distances as they are built.

To avoid complicating the translation of supercombinators into G-code by making it dependent on whether a supercombinator is traced or not, the instruction `MKTAP` is introduced by a subsequent peep-hole optimization pass.

4.3 EDT generation and the STG machine

The Spineless Tagless G-machine (Peyton Jones, 1992) as used in the Glasgow Haskell Compiler is the other popular abstract machine for compiled graph reduction. In principle, it is possible to integrate EDT-generating mechanisms as described in this section into the STG machine. An EDT-generating version of the operational semantics of the STG machine has been developed to verify this. We will not go into detail here, but the key points are as follows:

- The STG language needs to be extended with versions of `let` and `letrec` (called `letT` and `letrecT`) which conditionally build traced, updatable (potential) redexes, much like `MKTAP` does.
- Lambda forms need to be annotated with a flag indicating whether or not the tracer should be called when a closure for that lambda form is entered.
- Pieces of trace-related information need to be stored on the update and return stacks.
- When compiling for debugging, a special translation into the STG language must be used, employing `letT` and `letrecT` to build traced redexes on the heap.

5 Performance evaluation

5.1 Benchmarks and symbols

All measurements were performed on a 430 MHz Sun UltraSPARC equipped with 256 Mbyte of primary memory running Solaris. Six different benchmark programs were used, see below. All except Mini-Freja are from the NoFib suite (Partain, 1993). Due to the restrictions of the Haskell subset supported by the Freja compiler, the benchmarks from the NoFib suite had to be adapted (removal of user-defined class instances etc.). The benchmarks range from small to fairly large in terms of lines of source code (from 100 to 5800 lines, excluding comments and blanks), but all of them result in substantial computations (the execution times for non-debugged code ranged from 1.2 to 13 seconds, and between 1.2 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 is most important here.

- *Anna.* Strictness analyser. 5800 lines.
- *Cichelli.* Uses a brute-force search to construct a perfect hash function for a set of 16 keywords. 150 lines.
- *Clausify.* Transforms a proposition to an equivalent in clausal form. 100 lines.
- *Infer.* Type checker. 800 lines.
- *Parser.* Scans and parses 1760 lines of Haskell (the code for the parser itself repeated four times) and prints the resulting abstract syntax tree. 1200 lines.
- *Mini-Freja.* This is an interpreter for a small non-strict functional language. It interprets a program computing a list of the first 500 prime numbers using the sieve of Eratosthenes. 240 lines.

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, for example, garbage collection times were reduced.

Table 2 gives a breakdown of the execution time when the benchmark programs are compiled for ordinary execution with our system. To put the performance of our system into perspective, the column t_{HBC} gives the execution times for the benchmarks when compiled with HBC (Augustsson, 1997). Note that the garbage collection times in most cases are small as a result of having a heap which is much larger than the size of the live data. This was done to reduce the impact of the chosen garbage collection strategy for the base line cases and to make it possible to understand t_0 more or less as pure reduction time.

5.2 Debugging cost

This section evaluates the performance of our system when performing debugging. The six 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 for each benchmark.

The target 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 re-executions, it would be much larger than shown 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

Table 1. *Parameters and measured quantities.*

Symbol	**Parameter or measured quantity**
T	The number of *traced* reductions. This is a rough measure of the size of a computation. It is also 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. Observe 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 performing graph reduction.
t_{GC}	Garbage collection time. 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 a rough indication of the number of questions that can be answered before the target program is re-executed.

Table 2. *Breakdown of the execution time for the benchmark programs when compiled for ordinary execution. Average times over 5 runs.*

Benchmark	t_0 [s]	t_{red} [s]	$t_{\mathrm{GC}}{}^{a}$ [s]	t_{GC}/t_0	t_{HBC} [s]	t_{HBC}/t_0
Anna	2.5	2.4	0.1	0.03	2.7	1.1
Cichelli	4.1	3.7	0.4	0.09	3.4	0.8
Clausify	1.4	1.3	0.1	0.05	1.2	0.8
Infer	1.6	1.4	0.2	0.10	2.8	1.8
Parser	1.1	1.0	0.1	0.09	1.0	0.9
Mini-Freja	13.1	12.0	1.1	0.09	16.3	1.2

[a] The (initial) heap size was 16 Mbyte, except for Mini-Freja where 64 Mbyte was used.

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

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

Benchmark	*T*
Anna	3 194 501
Cichelli	5 550 908
Clausify	3 095 548
Infer	1 372 261
Parser	1 217 834
Mini-Freja	16 471 363

portion ($QD_{\max}$). This gives a (very) 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. The inconvenience of a possibly sluggish response also ought 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 to a large value[10] 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 and sometimes account for a significant part of the execution time.

Clausify is somewhat problematic since there are very large nodes (nodes which retain a lot of heap) 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 sometimes 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 peculiar entries for N and RG in the first three 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 (see 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 also grow, but again not as quickly.

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

Table 4. *Performance for different values of $RG_{\max}$. $N_{\max} = 1\ 000\ 000$.*

$RG_{\max}$ [Mbyte]	N [nodes]	RG [Mbyte]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\text{tot}}}{t_0}$	$\frac{t_{\text{red}}}{t_0}$	$\frac{t_{\text{GC}}}{t_0}$	$\frac{t_{\text{EC}}}{t_0}$
Anna								
1	37	0.0	78	10	4.0	1.3	2.0	0.7
2	37	0.0	78	12	4.6	1.3	2.5	0.8
4	11627	2.4	156	18	7.4	1.3	4.7	1.4
8	97437	2.9	205	27	10.8	1.3	7.1	2.4
16	996498	8.9	405	28	11.0	1.3	7.8	1.9
Cichelli								
1	6471	0.1	116	13	3.2	1.2	1.5	0.5
2	6936	0.2	117	17	4.0	1.2	2.0	0.8
4	259777	3.0	156	36	8.8	1.2	6.2	1.4
8	993945	5.4	215	66	16.2	1.2	13.0	2.0
16	993945	5.4	215	66	16.2	1.2	13.0	2.0
Clausify								
1	87	0.6	28	10	7.5	1.4	3.6	2.5
2	7	0.0	21	12	8.7	1.4	4.4	2.9
4	185	1.9	30	18	13.0	1.4	7.7	3.9
8	988201	5.8	59	24	17.4	1.4	12.8	3.2
16	988201	5.8	59	24	17.4	1.4	12.8	3.2
Infer								
1	43765	0.8	149	5.7	3.6	1.1	1.8	0.7
2	43765	0.8	149	6.5	4.2	1.1	2.2	0.9
4	43765	0.8	149	9.6	6.2	1.1	3.9	1.2
8	678815	8.9	596	13	8.3	1.1	5.4	1.8
16	999162	9.5	721	13	8.1	1.1	5.5	1.5
Parser								
1	201	2.2	98	5.5	5.2	1.2	2.9	1.1
2	224	2.2	99	8.3	7.9	1.2	5.1	1.6
4	79400	1.6	310	9.3	8.7	1.2	5.4	2.1
8	489836	9.9	624	15	14.6	1.2	10.2	3.2
16	979622	12.0	9999	15	14.2	1.2	10.2	2.8
Mini-Freja								
1	58542	0.7	418	31	2.3	1.3	0.6	0.4
2	58542	0.7	418	33	2.5	1.3	0.8	0.4
4	245107	2.0	836	33	2.5	1.3	0.8	0.4
8	999353	7.2	1669	48	3.7	1.3	2.1	0.3
16	999353	7.2	1669	48	3.7	1.3	2.1	0.3

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

$N_{\max}$ [nodes]	N [nodes]	RG [Mbyte]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\text{tot}}}{t_0}$	$\frac{t_{\text{red}}}{t_0}$	$\frac{t_{\text{GC}}}{t_0}$	$\frac{t_{\text{EC}}}{t_0}$
Anna								
5000	4513	1.8	148	4.2	1.7	1.3	0.3	0.1
10000	9841	2.3	154	4.4	1.8	1.3	0.3	0.2
50000	49957	2.7	182	5.8	2.3	1.3	0.7	0.3
100000	99640	3.1	206	7.0	2.8	1.3	1.2	0.3
500000	497714	5.8	321	30	11.8	1.3	9.2	1.3
Cichelli								
5000	4978	0.1	111	6.3	1.5	1.2	0.2	0.1
10000	9878	0.6	121	7.0	1.7	1.2	0.3	0.2
50000	46185	2.1	134	10	2.5	1.2	1.0	0.3
100000	92839	2.4	141	14	3.3	1.2	1.6	0.5
500000	497673	3.8	180	44	10.9	1.2	8.4	1.3
Clausify								
5000	4530	4.7	40	3.4	2.5	1.4	0.9	0.2
10000	7796	4.8	42	3.5	2.6	1.4	0.9	0.3
50000	36803	4.9	47	4.6	3.3	1.4	1.3	0.6
100000	96841	5.1	50	5.9	4.3	1.4	2.0	0.9
500000	430422	5.7	55	14	9.8	1.4	6.1	2.3
Infer								
5000	4852	0.1	71	2.0	1.3	1.1	0.1	0.1
10000	9868	0.2	82	2.0	1.3	1.1	0.1	0.1
50000	49622	1.4	163	3.1	2.0	1.1	0.5	0.4
100000	99626	3.6	248	6.4	4.1	1.1	2.0	1.0
500000	498621	7.9	519	10	6.7	1.1	4.0	1.6
Parser								
5000	4967	1.7	141	1.7	1.6	1.2	0.3	0.1
10000	9732	1.4	158	1.9	1.7	1.2	0.3	0.2
50000	49806	5.0	259	3.1	2.9	1.2	1.2	0.5
100000	99425	4.9	342	4.5	4.2	1.2	2.2	0.8
500000	499482	9.7	630	11	10.2	1.2	7.0	2.0
Mini-Freja								
5000	4930	0.4	135	21	1.6	1.3	0.1	0.2
10000	9952	0.4	183	21	1.6	1.3	0.1	0.2
50000	49841	0.7	387	22	1.7	1.3	0.2	0.2
100000	99710	1.0	540	23	1.7	1.3	0.3	0.1
500000	499217	3.8	1185	33	2.5	1.3	1.0	0.2

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

Benchmark	N [nodes]	RG [Mbyte]	$QD_{\max}$	t_{tot} [s]	$\frac{t_{\text{tot}}}{t_0}$	$\frac{t_{\text{red}}}{t_0}$	$\frac{t_{\text{GC}}}{t_0}$	$\frac{t_{\text{EC}}}{t_0}$
Anna	9481	2.3	154	4.4	1.8	1.3	0.3	0.2
Cichelli	9878	0.6	121	7.0	1.7	1.2	0.3	0.2
Clausify	7	0.0	21	3.1	2.3	1.4	0.6	0.3
Infer	9868	0.2	82	2.0	1.3	1.1	0.1	0.1
Parser	9732	1.4	158	1.9	1.8	1.2	0.4	0.2
Mini-Freja	9952	0.4	183	21	1.6	1.3	0.1	0.2

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 around a factor of 2 or better in all cases, while N and $QD_{\max}$ indicates that a reasonably large portion of the tree has been stored. The exception is again Clausify, where a somewhat overly aggressive pruning heuristic (see section 4.2.1) in this case made matters even worse. (4 Mbyte should have allowed at least 185 nodes to be kept; see table 4.) In fact, $N_{\max}$ was the limiting factor in all cases but the Clausify case.

In conclusion, these benchmarks show that the instrumentation overhead and the cost of building the EDT are reasonably low. (The true instrumentation overhead, i.e. when garbage collection is considered separately, was between 19 % and 46 % for these benchmarks.) 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 account for 70 % 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 collector 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 (Nilsson & Sparud, 1996). 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 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 (1995) 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. For instance, if the combined overhead of garbage collection and tree construction is roughly equal to the execution time of the target, then the speedup would be at most two.

Nevertheless, when debugging computationally intensive targets, re-execution is at some point going to be tedious. Unless the problem size can be made smaller, the only recourse would be to try to trace very selectively (using the trust and starting-point facilities) into a large trace store. However, it is worth keeping in mind that the problem of debugging computationally intensive applications is not unique to our setting. For instance, manual re-execution of a target is often necessary also during conventional debugging.

6 Related work

Until recently, the only readily available debugging tools for lazy functional languages were either low-level operational tracers (such as the tracing facilities offered by HBC (Augustsson, 1993)), 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 (Hazan & Morgan, 1993), or Sparud's stream programming debugger (Sparud, 1996)). This may now be about to change.

In order to construct general debuggers dealing with the intrinsic difficulties related to lazy evaluation, 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. (O'Donnell & Hall, 1988; Kamin, 1990)). However, what were proposed were in most cases *complete* traces, requiring storage in proportion to the size of the computation. The result is severe performance problems as soon as realistic programs are being traced.

Sparud (1994; 1996) 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) (Nilsson & Sparud, 1997).

The work by Naish and Barbour (1995) is closely related to Sparud's, and there are also similarities to the work presented in this article. 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 discussed in section 3.4, 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.

A prototype debugger based on the ideas of Naish and Barbour, and to some extent Sparud (1996), has been implemented and integrated into the Hugs interpreter (Pope, 1998). It uses a version of `dirt` which only makes use of features provided as standard with Hugs, thus avoiding any modifications of the underlying language implementation and making it portable across Hugs implementations. However, this debugger does not construct partial traces.

Recently, Sparud and Runciman have proposed an alternative debugging method based on maintaining complete computational histories, *redex trails*, for all values (Sparud & Runciman, 1997; Sparud, 1999). The system is known as Hat, the Haskell Abstract Tracer. The idea is that it should be possible to single out an erroneous value and follow its history *backwards* until the bug is found. 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. In many ways, a redex trail can be seen as an EDT with the edges reversed, except that it is much more detailed since every value (and not only redexes) is associated with its history.

Like Sparud's earlier work, Hat 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 (Sparud & Runciman, 1998). The employed scheme limits the length of redex trails by truncating trails at some fixed distance from a live piece of data or output fragment. In EDT terms, this is roughly as if pruning were to be done at some fixed distance above leaf nodes rather than at some distance below the root, reflecting the 'reversed' nature of a redex trail w.r.t. an EDT. The pruning 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. This would eliminate the risk of loosing important information. The direction of the redex trail edges makes storing a trail on file fairly straightforward, whereas re-execution for filling in missing parts of a trail seems difficult. This is the opposite to what is the case for an EDT.

An even more recent development is Hood, the Haskell Object Observation Debugger (Gill, 2000). Hood basically allows values (including functions, represented as partial mappings) to be observed by inserting what effectively amounts to `print`-statements in a target. However, the lazy evaluation order is unaffected, and it is even possible to observe how a value gradually becomes evaluated over the course of a computation. Like the other systems discussed here, Hood relies on tracing (to a file), and the size of the trace can cause performance problems. Hood is implemented as a Haskell library which only makes use of a few commonly implemented extensions of Haskell 98. Thus Hood is very portable across implementations, and it handles full Haskell to the extent that useful points for observation can be found in a target. Chitil, Runciman, and Wallace have performed an interesting comparison of Hat, Hood, and Freja (Chitil *et al.*, 2001).

7 Conclusions and future work

This article described the implementation of a debugger for lazy functional languages like Haskell. The central idea is the piecemeal construction of a declarative trace, the EDT, by repeated automatic re-execution of the target, the size of the pieces being determined by the amount of memory made available for debugging purposes. Given a few megabytes of trace storage and a reasonable limit on the number of stored EDT nodes, a traced program typically takes two to three times longer to execute than normal, while the stored portion of the trace is usually large enough to make re-execution a not too frequent event. The efficiency could probably be increased further by using a generational garbage collector instead of a copying one. One outstanding performance problem concerns the case when there are very large nodes close to the current EDT root. A solution might be pruning of argument and result *values* as a last resort.

The article also described what is needed to integrate our trace mechanisms into a language implementation based on graph reduction. Since substantial support from the compiler and run-time system is required, retrofitting a tracer as described herein to an existing system may not be trivial. On the other hand, it is not infeasible either, since the basic implementation principles assumed by the tracer are completely standard. In particular, as outlined in section 4.3, it is possible to integrate the trace mechanisms into the STG machine.

Some methods for reducing the size of the trace (trusted functions and starting points) were also outlined. There are room for further improvements in this respect. For instance, when a program terminates abnormally, it is often useful to start debugging 'close' to the error, just as one would do in a debugger for an imperative language. This could be achieved by recording the *complete* child-parent relation, i.e. the skeleton of an inverted EDT, which seems easy since the needed information, the parent identity, is available in traced application nodes. The resulting structure would be similar to what Hazan and Morgan proposed (Hazan & Morgan, 1993), or to a heavily pruned redex trail (Sparud & Runciman, 1998), allowing debugging to be started anywhere along the syntactic call chain.

Having some form of support for monadic programming, and maybe also for various forms of combinator style programming, would be very desirable. The problem is that monad/combinator structured code currently only can be debugged at the implementation level of the abstraction in question. This is often inconvenient. Unfortunately it is not clear how to address this problem. Some kind of user-extensible mechanisms are probably called for.

It might also be useful to have the option of tracing at a more fine-grained level, e.g. local variable bindings, guards, and `case`-expressions. Note that redex-trails provide tracing at that level of granularity (Sparud & Runciman, 1997). This can be achieved by making the complier insert extra lambda-abstractions at suitable points. We have already described how to handle list comprehensions in that way (Nilsson, 1998).

Finally, the techniques need to be extended to handle full Haskell (at least adequate support for I/O) as well as commonly implemented extensions such as unsafe execution of I/O computations, concurrency, and imperative state. It might be that a more traditional type of debugging would be appropriate at this level, and thus one would have to find ways of integrating such functionality into our declarative debugging framework. In a way, this would just reflect the architecture of a language like Haskell, with its imperative-looking, top-level I/O layer on the one hand, and the purely functional foundation on the other.

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 (1987), 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.

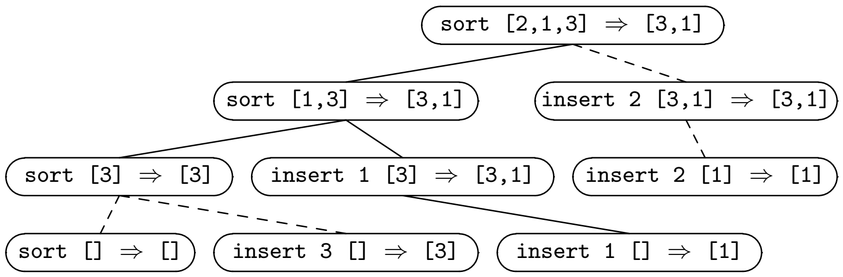
The Freja debugger is basically an algorithmic debugger, even though it is easy enough to use it more or less as a conventional debugger to explore a computation, should that be desired. Algorithmic debugging (Shapiro, 1982), originally developed for logic programming, is a semi-automatic debugging technique where the debugger tries to locate the node which is ultimately responsible for a visible bug symptom in

```
insert x []                   = [x]
insert x (y:ys) | y > x       = y : (insert x ys)
                | x < y       = x : y : ys
                | otherwise   = y : ys

sort []     = []
sort (x:xs) = insert x (sort xs)

main = sort [2, 1, 3]
```

(a) Erroneous insertion sort program.



(b) The EDT for the insertion sort program. Explored paths in the EDT are solid, unexplored paths dashed.

```
sort [2,1,3]  ⇒ [3,1] ?
> no
sort [1,3]  ⇒ [3,1] ?
> no
sort [3]  ⇒ [3] ?
> yes
insert 1 [3]  ⇒ [3,1] ?
> no
insert 1 []  ⇒ [1] ?
> yes
Bug located in function "insert".
Erroneous reduction: insert 1 [3]  ⇒ [3,1]
```

(c) Algorithmic debugging of the insertion sort program.

Fig. 12. Algorithmic debugging illustrated.

an execution tree such as an EDT. In a functional setting, this is done by checking whether the recorded reductions are correct or not, typically in a top-down order, by asking the user or by referring to some formal specification (Nilsson & Fritzson, 1994). The search ends once a node is found which itself represents an erroneous reduction but whose children all represent correct computations. It can then be

Table 7. *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.*

Productions[a]			**Attribute equations**[b]		
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_R	$T_\mathrm{R}{\downarrow}itips$	=	$T{\downarrow}itips$
			$T_\mathrm{L}{\downarrow}itips$	=	$T_\mathrm{R}{\uparrow}stips$
			$T{\uparrow}stips$	=	$T_\mathrm{L}{\uparrow}stips$
			$T_\mathrm{L}{\downarrow}isorted$	=	$T{\downarrow}isorted$
			$T_\mathrm{R}{\downarrow}isorted$	=	$T_\mathrm{L}{\uparrow}ssorted$
			$T{\uparrow}ssorted$	=	$T_\mathrm{R}{\uparrow}ssorted$
			$T{\uparrow}tree$	=	$T_\mathrm{L}{\uparrow}tree$ `:^:` $T_\mathrm{R}{\uparrow}tree$

[a] S is the start symbol.
[b] $\downarrow$ indicates an inherited attribute, $\uparrow$ a synthesized one.

concluded that the function being applied in the located node contains at least one bug. The process is illustrated in figure 12.

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 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, starting the development from an attribute grammar can be helpful. The grammar can 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 table 7. Figure 13 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 non-terminal. The functions are defined by pattern-matching over the tree type. There is one case for each of the non-terminals' 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

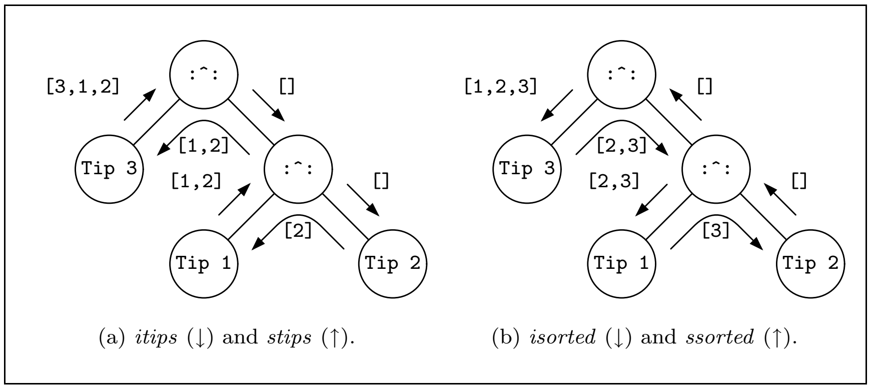

[3,1,2]
[]
[1,2]
[1,2]
[]
[2]
:^:
Tip 3
Tip 1
Tip 2
(a) *itips* (↓) and *stips* (↑).
[1,2,3]
[]
[2,3]
[2,3]
[]
[3]
(b) *isorted* (↓) and *ssorted* (↑).

Fig. 13. Attribute propagation for a small tree.

in figure 14. However, the transliteration was performed rather carelessly, resulting in a mistake.

When the program is executed, it immediately stops with an error message saying that a black hole has been encountered: `[Fatal error] Black hole!` Since this does not offer any particularly good lead as to what the problem might be, we recompile the program with debugging support and start it in debug mode. Below, the user's input is typeset in italics.

```
FREJA DEBUGGER
--------------
[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 _|_) ?) ?
```

```
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)
```

Fig. 14. 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 7, but contains a bug.

```
3> no
```

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. 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* (see section 2.1) 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.

```
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 $\perp$. The black hole appears as the second component of the returned tuple, i.e. `t_ssorted` is bound to $\perp$. `t_ssorted` is equal to `r_ssorted` which depends on `r_isorted` via the definition of `(sortTree' (Tip 5))` (instantiation of the definition gives `r_ssorted = tail r_isorted`). 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

Augustsson, Lennart. 1984 (Aug.). A compiler for Lazy ML. *Pages 218–227 of: Proceedings of the 1984 ACM conference on LISP and functional programming.*

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

Augustsson, Lennart. (1997). *The HBC compiler.* `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`

Chitil, Olaf, Runciman, Colin, & Wallace, Malcolm. (2001). Freja, Hat and Hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. *Pages 176–193 of:* Mohnen, Markus, & Koopman, Pieter (eds), *Proceedings of the 12th international workshop on implementation of functional languages (IFL 2000), aachen, germany, september 2000.* Lecture Notes in Computer Science, vol. 2011. Springer-Verlag.

Gill, Andy. (2000). Debugging Haskell by observing intermediate data structures. *Proceedings of the 2000 ACM SIGPLAN haskell workshop, montreal, canada.* Nottingham University.

Hazan, Jonathan E., & Morgan, Richard G. 1993 (May). The location of errors in functional programs. *Pages 135–152 of:* Fritzson, Peter (ed), *Automated and algorithmic debugging.* Lecture Notes in Computer Science, vol. 749.

Johnsson, Thomas. 1984 (June). Efficient compilation of lazy evaluation. *Pages 58–69 of: Proceedings of the 1984 ACM SIGPLAN symposium on compiler construction.* Proceedings published in ACM SIGPLAN Notices, 19(6).

Johnsson, Thomas. (1987). Attribute grammars as a functional programming paradigm. *Pages 154–173 of: Functional programming languages and computer architecture.* Lecture Notes in Computer Science, vol. 274. Portland, Oregon: Springer-Verlag.

Kamin, Samuel. 1990 (July). *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.

Morris, J. H. (1982). Real programming in functional languages. Darlington, J., Henderson, P., & Turner, D. A. (eds), *Functional programming and its applications*. Cambridge University Press.

Naish, Lee, & Barbour, Tim. (1995). *Towards a portable lazy functional declarative debugger*. Technical Report 95/27. Department of Computer Science, University of Melbourne, Australia.

Nilsson, Henrik. 1998 (May). *Declarative debugging for lazy functional languages*. PhD thesis, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden.
`http://www.ida.liu.se/~henni/thesis.ps`

Nilsson, Henrik. (1999). Tracing piece by piece: affordable debugging for lazy functional languages. *Pages 36–47 of: Proceedings of the 1999 ACM SIGPLAN international conference on functional programming*. Paris, France: ACM Press.

Nilsson, Henrik, & Fritzson, Peter. (1994). Algorithmic debugging for lazy functional languages. *Journal of functional programming*, **4**(3), 337–370.

Nilsson, Henrik, & Sparud, Jan. 1996 (Aug.). *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. This is an extended version of (Nilsson & Sparud, 1997).

Nilsson, Henrik, & Sparud, Jan. (1997). The evaluation dependence tree as a basis for lazy functional debugging. *Automated software engineering*, **4**(2), 121–150.

O'Donnell, John T., & Hall, Cordelia V. (1988). Debugging in applicative languages. *Lisp and symbolic computation*, **1**(2), 113–145.

Partain, William. (1993). The NoFib benchmark suite of Haskell programs. *Pages 195–202 of:* Launchbury, John, & Sansom, Patrick (eds), *Proc. 1992 glasgow workshop on functional programming*. Workshops in Computing. Springer-Verlag.

Peyton Jones, Simon L. (1987). *The implementation of functional programming languages*. Prentice Hall.

Peyton Jones, Simon L. (1992). Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of functional programming*, **2**(2), 127–202.

Pope, Bernard. 1998 (June). *Buddha: A declarative debugger for Haskell*. Honours thesis, Department of Computer Science, University of Melbourne, Australia.

Runciman, Colin, & Röjemo, Niklas. (1996). New dimensions in heap profiling. *Journal of functional programming*, **6**(4), 587–620.

Runciman, Colin, & Wakeling, David. (1993). Heap profiling of lazy functional programs. *Journal of functional programming*, **3**(2), 217–245.

Sansom, Patrick M., & Peyton Jones, Simon L. (1995). Time and space profiling for non-strict higher-order functional languages. *Pages 355–366 of: Principles of programming languages (POPL '95)*.

Shapiro, Ehud Y. (1982). *Algorithmic program debugging*. MIT Press.

Sparud, Jan. 1994 (Jan.). *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.

Sparud, Jan. 1996 (Feb.). *A transformational approach to debugging lazy functional programs*. Licentiate thesis, Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden.

Sparud, Jan. 1999 (Mar.). *Tracing and debugging lazy functional computations.* PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96, Göteborg, Sweden.

Sparud, Jan, & Runciman, Colin. 1997 (Sept.). Tracing lazy functional computations using redex trails. *Proceedings of the 9th international symposium on programming languages, implementations, logics and programs (PLILP '97).*

Sparud, Jan, & Runciman, Colin. (1998). Complete and partial redex trails of functional computations. *Pages 160–177 of:* Clack, Chris, Hammond, Kevin, & Davie, Tony (eds), *Proceedings of the 9th international workshop on implementation of functional languages (IFL '97), st. andrews, scotland, september 1997.* Lecture Notes in Computer Science, vol. 1467. Springer-Verlag.

Wadler, Philip. (1998). Why no one uses functional languages. *ACM SIGPLAN notices*, **33**(8), 23–27.