Complete and Partial Redex Trails of Functional Computations

Jan Sparud and Colin Runciman

Department of Computer Science, University of York, UK
(e-mail: {sparud,colin}@cs.york.ac.uk)

Abstract. Redex trails are histories of functional computations by graph reduction; their main application is fault-tracing. A prototype implementation of a tracer based on redex trails [8] demonstrated the promise of the technique, but was limited in two respects: (1) trails did not record every reduction, only those constructing a new value; (2) even so computing trails was very expensive, particularly in terms of the memory space they occupied. In this paper, we address both problems: *complete* redex trails provide a full computational record; *partial* versions of these trails exclude all but selected details, greatly reducing memory costs. We include results of experiments tracing several applications, including a compiler.

1 Introduction

Programs do not always work first time. When they fail, programmers cannot always see the reason immediately. Implementors must therefore provide some way of *tracing* computations. By examining traces, programmers can see not only final results or failures, but also how these have come about.

In many programming languages, traces can be based on sequences of events. But in a lazy functional language, the sequence of events is not so easily presented, nor is it what the programmer really needs. A more appropriate form of trace links a final value, or failure, to its *parent redex*, and each part of that redex in turn to *its* parent, and so on until one reaches input data or initial expressions present in the program. We call such traces *redex trails*. In a previous paper [8] we described a method for transforming functional programs to self-tracing equivalents, computing not only the result as usual but also a redex trail showing how the result was obtained. In addition, we described a *trace browser* used to examine traces in close connection with programs and their results. The scheme was promising, but its scope of usefulness was limited in two respects.

First, not every reduction was recorded in a redex trail, only those with the construction of a new value as the outcome. For some applications we found this was enough to provide highly informative traces. But in others, for example those with a strong search component, large parts of the computation are concerned with conditions for selecting between given values; if details of these parts are not recorded in trails, the programmer examining traces lacks important information.

Secondly, and despite the restricted class of reductions recorded, it was very expensive to compute redex trails. For a range of small applications we found that a trace-constructing computation could take 30 times longer than a normal one. The demands for memory space were even more prohibitive, with each recorded reduction taking up to 100 bytes. Tracing was just not feasible for computations beyond about a million reductions — sufficient for most textbook exercises, but not for serious applications.

In this paper we describe how we have tackled both problems. Redex trails have been extended to include a representation for every reduction that occurs, providing complete information about all parts of a computation. Desirable as this may be, it makes trails even larger, and more expensive. We have implemented two techniques for controlling the construction and retention of redex trails, so that stored trace structures are simpler and smaller: it is indeed useful to confine trails to a limited class of reductions, but now this class is *selected* by the programmer in terms of the program components they wish to observed at work; if demands on memory are still too high, we *prune* redex trails using harsh garbage-collection policies.

We give results for a range of applications, including two much larger than those studied in the previous paper.

Section 2 briefly explains the main elements of redex trails, as implemented in our prototype tracing system. Section 3 identifies the class of reductions not recorded by that system, and describes the extension to *complete* redex trails. Section 4 is concerned with *partial trails*; it deals first with *selective* trails and then with the complementary technique of trail *pruning*. Section 5 gives results, concentrating on the space requirements of the enhanced tracing system. Section 6 discusses related work. Section 7 draws conclusions and suggests future work.

2 A Redex-Trail Tracing System

Suppose the function `length` is defined by the equations

```
length [] = 0
length (x:xs) = 1 + length xs
```

and we compute `3` as the result of `length [1,2,3]`. Figure 1 shows a trace of that computation in the form of a *redex trail*. The *parent redex* of the final `3` is `1+2`. Considering subexpressions in that redex, everything but the `2` was constructed as an immediate result of reducing the original expression `length [1,2,3]`, but the `2` came about by reduction of `1+1`. And so on.

As described in a previous paper [8], we have modified the Haskell compiler `nhc` [5, 6] to provide traces based on redex trails. On request, our modified compiler applies a program transformation causing each value in the compiled program to be computed along with a redex trail recording the value's origin. A special browser, illustrated in Figure 2, allows the user to examine redex trails, starting from program output, or a run-time fault, or some point at which evaluation was interrupted.

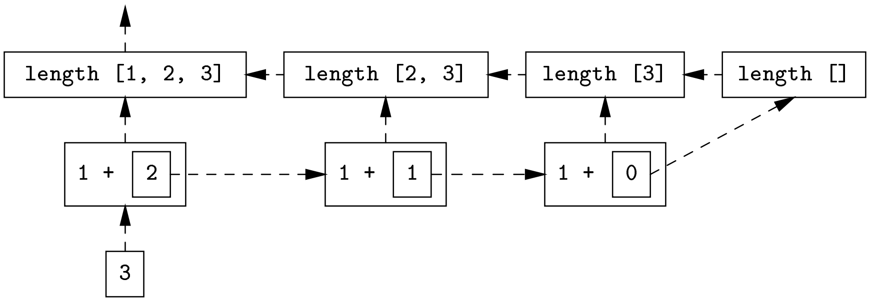
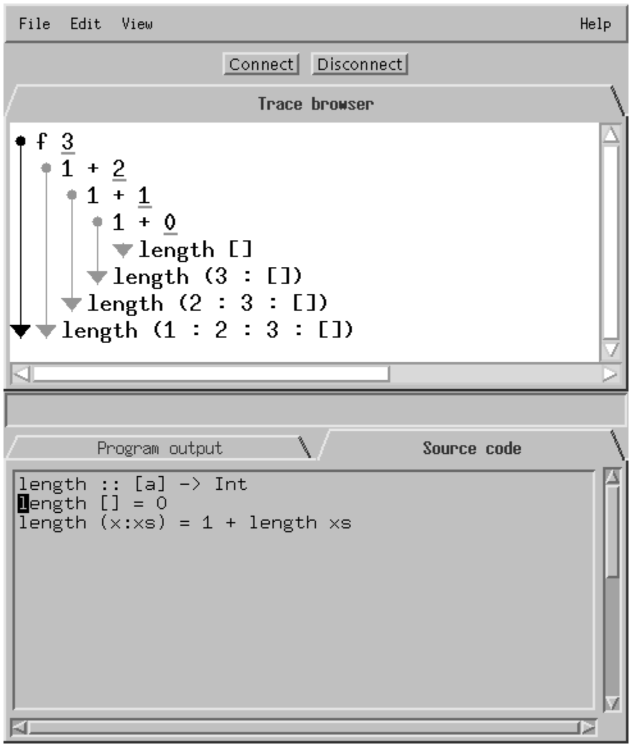


Fig. 1. The full redex trail for the computation of `3` as the result of `length [1,2,3]`.



```
f 3
 1 + 2
  1 + 1
   1 + 0
    length []
   length (3 : [])
  length (2 : 3 : [])
 length (1 : 2 : 3 : [])
```

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Fig. 2. How redex trails are displayed by the trace browser, shown here being used to examine the `length` trail of Figure 1. Each redex on display has been requested by the user in a point-and-click style. Available links to the source program include the points of definition and use for each variable occurring in the trail, and also the site of each function application.

Redex trails are themselves implemented as Haskell data structures of type `Trace`. A simplified definition of the `Trace` datatype is:

```
data Trace = Appl Trace [Trace] SourceRef
           | Name Trace NameInfo SourceRef
           | Root
```

Each construction `Appl` t_c `[`t_f`,` t_{a_1}`,` ... `,`t_{a_n}`]` *sr* records a function application. The trail recording the origin of the function is t_f, and the trails t_{a_i} are for its arguments. The trail t_c, which we term the *application context* records how the application of this function to these arguments came about. (In a first-order language t_c and t_f would always be the same, but in a higher-order language it is important to distinguish between them.) The source reference *sr* allows immediate access to the corresponding function application in the source program.

Each construction `Name` t i *sr* represents an identifier, constant or literal. Its redex trail is t. The information field i records a textual representation of the name, and the module and position of its *definition*. The source reference *sr* is linked to the relevant applied occurrence or *use* of the name.

The `Root` constructor is used as a trail terminator: for example, as the trail of each name with a top-level definition.

As all values in a traced computation, including functions, have trails attached to them, traced function applications are translated by the compiler to applications of special combinators that extend the redex trails appropriately.

3 Complete Trails

In the previous section we had a good reason for choosing the `length` function as an example: our prototype system builds *complete* redex trails for `length` computations, including a record of every reduction. But now consider another simple function over lists:

```
last [x] = x
last (x:xs) = last xs
```

Suppose we again compute `3`, but this time as the result of `last [1,2,3]`. In contrast to Figure 1, the trail constructed for this computation by our prototype tracer is not very informative (see Figure 3).

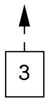


Fig. 3. A rather uninformative redex trail for the result of `last [1,2,3]`.

A programmer examining this trail to answer the question 'Where does the result `3` come from?' finds only the answer that `3` was given in the original expression; there is no mention of the `last` function or its workings. Why such a dramatic difference between the trails for `length` and `last` computations? Because a trail attached to a value by our prototype system records only how that value itself came to be computed. Shared values have identical trails. If the result of a function is (a component of) one of its arguments the trail for the result is exactly the trail already recorded in the argument.

Results that are not fresh values also pose a problem for graph-reducing interpreters or compilers: the application subgraph must somehow be updated to become, in effect, a piece of graph already constructed elsewhere. A commonly adopted solution is to introduce an indirection. We also use indirections to record such reductions in extended redex trails, giving `Trace` a new construction:.

```
data Trace = ... | Indi Trace Trace
```

A trace of the form `Indi` t_{res} t_{app} is constructed for the result of an indirecting application trace t_{app}, the result itself having trace t_{res}. The revised trail for the `last [1,2,3]` example is shown in Figure 4.

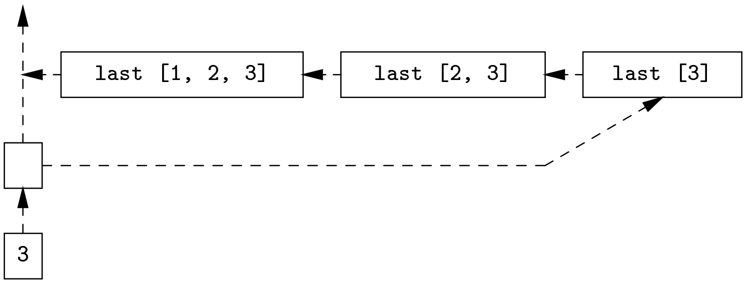


Fig. 4. A complete redex trail for `last [1,2,3]`. The empty box represents an indirection giving access to the chain of recursive calls selecting `3` as the final list element as well as to the trace of `3` itself.

With the addition of these indirection structures, redex trails become complete. Every reduction occurring in the computation can be observed in the trail.

4 Partial Trails

Our prototype tracer needed tens of megabytes to store the trails of computations involving a few hundred thousand reductions. Plenty of functional programs take many millions of reductions to compute their results. And with redex trails extended to include all reductions, not just the value-producing ones, the demand for memory will be even greater. There is a clear need for techniques to obtain simplified, smaller variants of a full redex trail.

4.1 Selective Trails

Not even the most inquiring programmer will be able (or willing!) to look at every single reduction in a large computation. So why record all of them? The problem is how to know which reductions to record and which to ignore. One solution is to let the programmer select the parts to be traced when they run a program.

Suppose each function defined at the top level of a program is either *suspected* or *trusted* by the programmer, and somehow marked as such. (Even in a freshly written program, obvious candidates for trusted functions include any from the `Prelude`.) Broadly speaking, an application of a suspected function should always be recorded in the redex trail, but for trusted functions this may be unnecessary.

For example, when taking the length of a list, we are rarely interested in how `Prelude.length` works, so recording a full redex trail for the internal computation of `length` would be excessive, both from the programmer's point of view and in terms of memory space. If we trust `length` (and primitive addition), then instead of the full redex trail in Figure 1 we might prefer the simpler *partial trail* shown in Figure 5. Note that the top-level call to `length` is still in the trace. Though we trust the internal workings of `length`, we still want to see the *original application* that reduced to `3`.

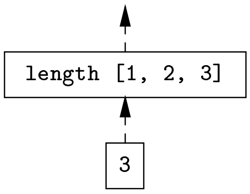


Fig. 5. A selective trail for `length [1, 2, 3]` when `length` is trusted.

The tracing machinery has to decide for each function application whether or not to record it in the redex trail. A simple rule would be: *Do not record applications in a trusted context.* This would produce the desired result for the `length` function, as all the intermediate applications of `length` and `+` occur in the context of `length` itself.

Higher-order Functions However, matters get more complicated when we decide to trust a higher-order function. Figure 6 shows a full redex trail from the result of `map dbl [1, 2, 3]`, where `dbl` is a function that doubles its argument. Suppose we trust `map` but not `dbl`. Should we then trust the result of `map dbl [1, 2, 3]`? Clearly not. The problem is that the trusted function `map` builds applications of the suspected function `dbl`. If we follow our simple rule, we will not record applications of the suspected function `dbl`, as figure 7 shows.

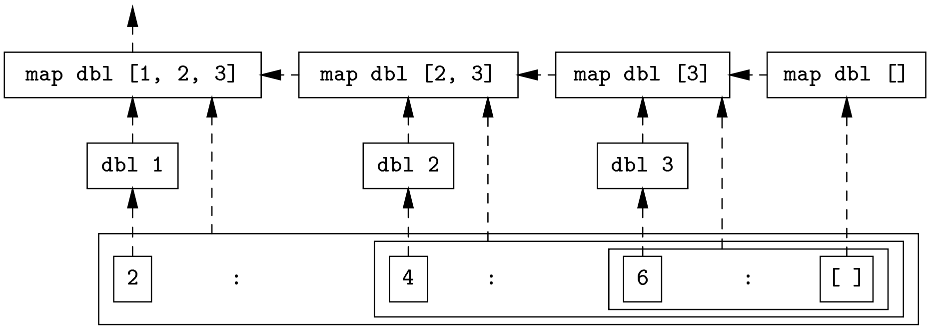


Fig. 6. The complete trail for `map dbl [1, 2, 3]`.

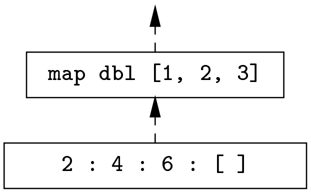


Fig. 7. A selective trail for `map dbl [1, 2, 3]` when `map` is trusted but `dbl` is suspected, if context alone determines which reductions are recorded.

To solve this problem we revise the rule: *Do not record applications of trusted functions in trusted contexts.* Figure 8 shows the effect of this revised rule for the `map dbl` example: there is no record of applications within `map`, except for the calls to `dbl`.

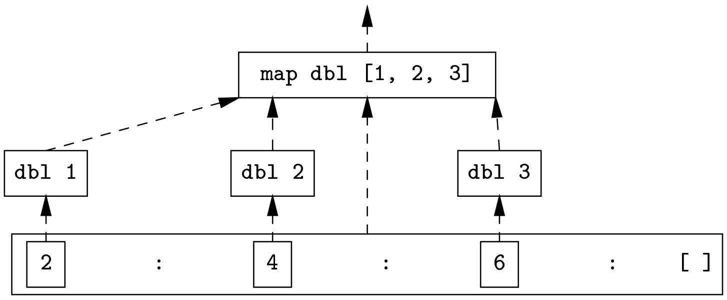


Fig. 8. The trace of `map dbl [1, 2, 3]` when `map` is trusted and `dbl` is suspected, using both the context and the applied function to determine recorded reductions.

Implementing Selective Trails To implement selective trails, we need to be able to find out if the context and the applied function are trusted when a function is applied. We can find out the name of the function in the current context and the name of the function to be applied by inspecting their accompanying trace structures. But as this operation will be performed every time a function is applied, it is important to make it cheap. The combinators for traced function application are written in Haskell, but to use a Haskell data structure to record whether a function is trusted or not is inappropriate for two reasons: (1) it would be hard to create the structure in the context of separate compilation; (2) it would be expensive to lookup the information in it.

Recall that wherever a name occurs in a redex trail, there is a `Name` construction in the `Trace`, including a record of all needed information about the name. The record for each name is unique, shared across all `Name` occurrences. For each module, the compiler constructs a table of records for all the names the module defines. Also, for each module there is a table of imported modules; so from the main module it is possible to reach the name table for all modules in the program.

So we simply add a 'trusted' flag to every name record. Whenever a function is applied, the application operator checks if the trusted flag is set for *both* the context of the application *and* the function being applied. The redex trail is only extended if this test fails.

The programmer can decide at compile-time whether or not to trust all the functions in a module by default. But run-time flags allow the default to be overridden. The implementation is a straightforward search through import tables, starting from the main module, to find the name table of the specified module. There the trusted flags are set or reset as needed. The search is performed once only, before starting the functional computation.

4.2 Trail Pruning

Selective trails should go some way to solving the problem of excessive memory demands. But even with reductions within the *Prelude* and other low-level modules excluded, a redex trail for something like a Haskell compilation may be so large as to be impractical. If tracing is to be feasible even for large computations, we need something more.

Another way of limiting the size of redex trails is to apply some kind of *pruning* when memory starts to run out. This idea prompts two questions: (1) when to prune? and (2) what to prune? A reasonable answer to the first question is to prune redex trails during each garbage collection. The garbage collector is called when available heap memory is exhausted; it already scans the heap including both normal program data and trails. Pruning can be implemented as a harsh garbage collection policy. The second question is harder. Our approach is simple: limit the *length* of all trails accessible from live program data or output. Trails beyond a certain length k are pruned, by overwriting them with a new `Trace` constructor `Pruned`.

When exploring a trace that has been pruned to length k, the user is guaranteed to be able to follow each trail at least k steps. When reaching a pruned part of the trace, the display program indicates that the information is no longer available. For the common error that no clause in a function definition matches

Table 1. A comparison of memory costs for incomplete redex trails without indirecting applications and complete trails with them.

	Incomplete	Complete
`cichelli`	13 Mb	14 Mb
`clausify`	28 Mb	28 Mb
`primes`	54 Mb	85 Mb

the actual argument (eg. `head []`), setting $k = 1$ is enough to be able to find out which function called the non-matching function (and the source position of the call). With $k = 2$, the top level of the argument will still be accessible.

5 Experimental Results

What are the practical consequences of extending the redex-trail tracer in the ways described in previous sections? To find out we first looked again at the three benchmark computations studied in our earlier paper: `cichelli` uses a brute-force search to construct a perfect hash function for a set of 16 keywords; `clausify` transforms a proposition to an equivalent in clausal form; `primes` computes the first 2,500 primes using a wheel sieve.

5.1 Extra Cost of Complete Trails

How much larger are the complete trails, including indirecting applications, than trails without them? The comparative memory costs are shown in Table 1. Recall our experience with the prototype: we felt a lack of information about some applications but not others. For `primes`, completing the trail means increasing its size by over 50%, but for `cichelli` and `clausify` the trail sizes hardly change.

5.2 Sizes of Selective Trails

Our first remark about the practical consequences of selective trails is a subjective one. We immediately appreciated the benefit of suppressing detail within the `Prelude` when examining traces.

More objectively, how much smaller are selective trails than full ones? As Table 2 shows, when `Prelude` functions are trusted, between 40% and 80% of reductions recorded in the full trails of these programs are omitted in a selective one. There is a corresponding saving in memory, but not necessarily in proportion. Some function applications have far larger representations than others depending, for example, on the number and size of arguments and whether these are shared. For `cichelli` and `clausify` the memory saving is only 20%–30% — rather less than the reduction figures might lead one to expect. For `primes`, on the other hand, the fraction of memory saved far exceeds such expectations:

Table 2. The effect of excluding trace information within a trusted `Prelude`.

	Full Trail	Selective Trail		No Trail
	#reductions	#reductions	memory	memory
`cichelli`	355 k	213 k (60%)	10.0 Mb	34 kb
`clausify`	1406 k	291 k (21%)	23.2 Mb	7 kb
`primes`	981 k	322 k (33%)	0.8 Mb	60 kb

about a third of all reductions are still recorded, yet the amount of memory needed shrinks by almost two orders of magnitude!

However, in all cases the final column of Table 2 provides a salutary reminder of how large the trails still are compared with the structures in memory for the normal untraced computation. Even selective redex trails can increase heap-memory requirements by a huge factor. Such increases may be infeasible for larger applications, confirming the need for a more drastic space-saving technique such as pruning.

5.3 How Effective is Pruning?

To test the effectiveness of pruning we again constructed traces for the three benchmark computations `cichelli`, `clausify` and `primes`, but with trails beyond some length k pruned away at each garbage collection. Figures 9 and 10 confirm the expected result that pruning with small values of k can save a great deal of space. What the figures do *not* show is that in practice even for very small values of k one finds a good deal of information in traces — after all, the last garbage collection may have been some time ago. Even setting $k = 0$ is a worthwhile option, and the memory overhead in that case is only a small constant multiple of normal memory use.

Another observation from these graphs is that the fraction of memory that is saved for a given pruning length k varies widely between applications. Compare the effect of pruning the `clausify` trail (upper part of Figure 10) with the effect of pruning the `primes` trail (lower part of Figure 10). For `clausify`, even 'light pruning' with $k = 16$ is enough to remove about 95% of the trace: we have had to omit the plot for $k = \infty$, to make comparisons between other k values visible! For `primes`, however, the memory saved by pruning with $k = 16$ is only 1%–2%, barely discernible in the graph, and much harsher pruning is needed to make appreciable memory savings.

Any pruning scheme based on a fixed numeric limit has the drawback that the user may be unsure what limiting value is appropriate. An advantage of directly limiting path-length is that the user knows what losses to expect when it comes to browsing the trace. But what the user may *not* know is whether the trace will fit in available heap memory. As an alternative to fixing the value of k, programmers could specify only an overall bound on the size of heap memory. The implementation would then vary k automatically to achieve (only) the necessary degree of pruning.

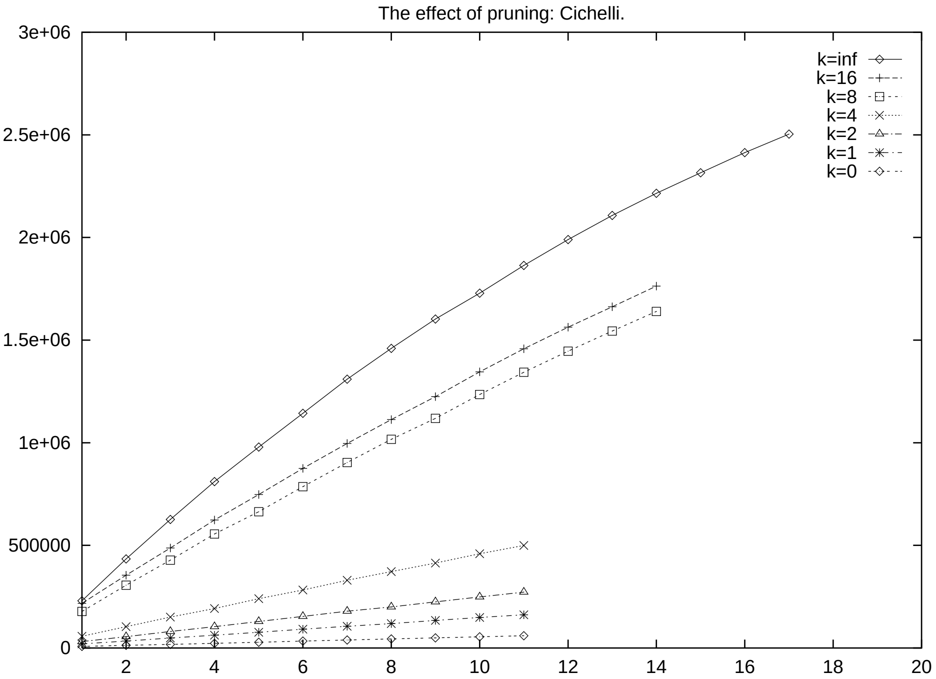


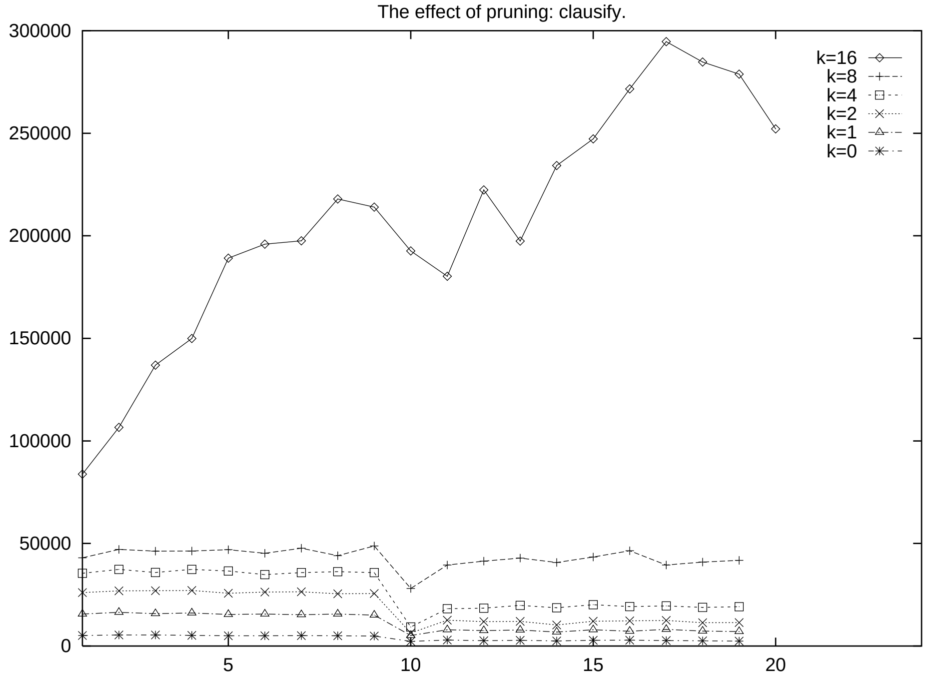
Fig. 9. The effect of pruning trails for the `Cichelli` computation beyond a fixed length k at each garbage collection. The x axis shows the number of garbage collections, and the y axis the amount of live heap memory in words.

5.4 Larger Applications

So far we have looked at three quite different but modest benchmark computations, in which programs of up to a few pages in length perform up to a million reductions. Completing redex trails with a record of indirecting applications has not caused a blow-up in memory demands. Partial trails require far less memory than full trails and are still effective as sources of information.

One motivation for partial trails is the wish to handle more demanding applications, beyond the capacity of our prototype system. We now give results for two such applications.

A Chess End-game: Wurzburg-`mate` Our first larger application involves brute-force search to solve a chess end-game problem of the *White to move and mate in N* variety. The program, called `mate`, is not so very large — about 400 lines excluding `Prelude` definitions. But the problem we shall supply as input (shown in Figure 11) is sufficiently tough that the computation requires over 20 M reductions.



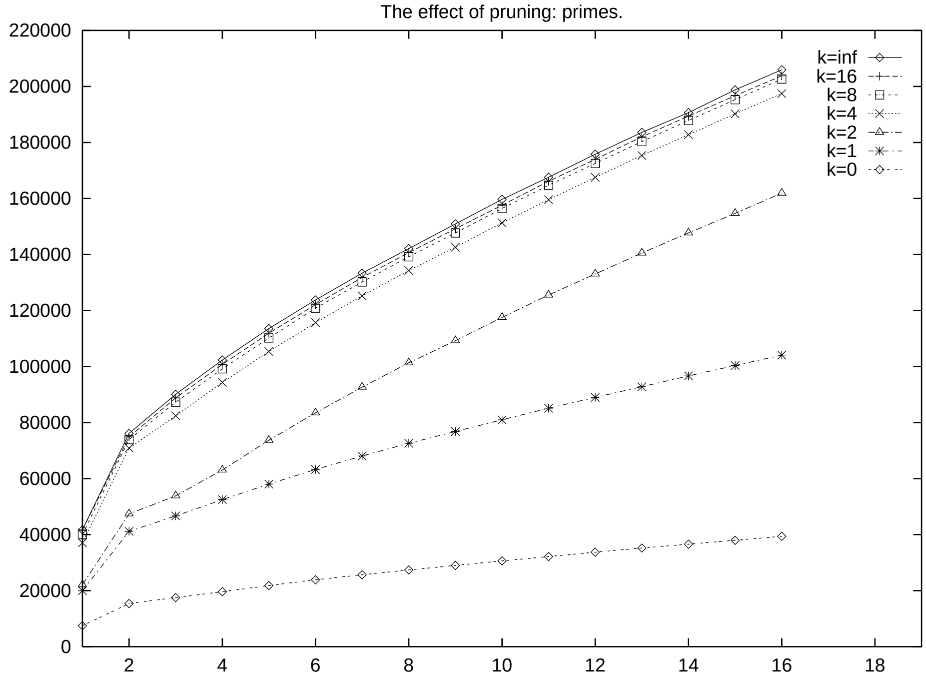


Fig. 10. The effect of pruning trails for the `clausify` and `primes` computations. Again the x axis shows the number of garbage collections, and the y axis the amount of live heap memory in words.

```
-- O. Wurzburg, Pittsburgh Gazette-Times, 1914
-- The classic pawn-promotion three-mover.

- - k - - - - -
r - - - - - - -
- - - p K - - -
- - - - P - - -
- - - p p b - -
- - - p - p - -
- - - - - - - -
- - - - - - - -

White to play and mate in 3
```

Fig. 11. The problem used as input to the `mate` program. “A striking task achievement presented with due regard to artistic canons.” [4]. Chess-playing readers might like to verify that solving it requires a significant amount of computation!

As one would expect, the `mate` computation is dominated by functions testing and sifting board positions and moves. The results of many of these functions are not fresh constructions but components of arguments. So trails produced *without* indirections are far from complete: constructing a selective trail of that kind with the `Prelude` trusted requires no more than 0.5 Mb memory space. However, some of the most interesting aspects of the computation are not recorded!

So how much space is required to record the Wurzburg-mate computation in a redex trail including the so-called indirecting applications? Without some selectivity, building the trail in heap memory is not feasible — by extrapolation, it would require over 100 Mb. Table 3 shows the extent of various selective trails. Apart from the `Prelude`, the lowest level module in `mate` defines a `Board` type, and basic operations on it such as vacating a square, or occupying one with a given piece. At the next level up, the `Problem` module parses a problem statement into an initial state, and the `Move` module defines legal moves.

Table 3. The magnitude of redex trails for the Wurzburg-`mate` computation, with varying degrees of selectivity.

trusted modules	# reductions in trail	memory needed
none	20,344 k	unknown
`Prelude`	9,827 k	52.7 Mb
`Prelude,Board`	8,371 k	51.2 Mb
`Prelude,Board,Move,Problem`	2,571 k	21.4 Mb

Even the most selective trail boosts the memory requirement from less than 10 kb for an untraced (but deliberately space-efficient) search to over 20 Mb

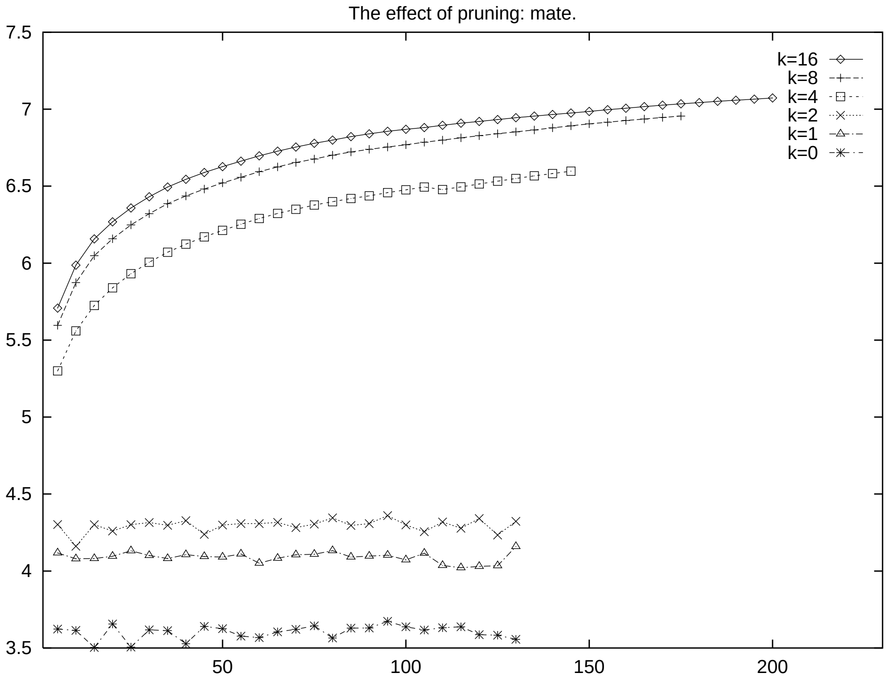


Fig. 12. The effect for the Wurzburg-`mate` computation of pruning trail length to k at each garbage collection, for various values of k between 0 and 16. Only the `Prelude` module is trusted. The y-axis now uses a *log scale* with base 10: each plotted point (x, y) represents 10^y words of live heap memory recorded after x garbage collections.

— giving some insight into the number of moves and positions the 'suspected' `Solution` module deals with. Also, without tracing the program runs for about two and a half minutes, but with selective trail construction this increases to between 15 and 30 minutes.

Figure 12 shows the effect for Wurzburg-`mate` of increasingly severe trail-pruning at each garbage collection. Light pruning ($k \geq 4$) is not enough: memory demands still grow steadily to tens of megabytes. But setting $k \leq 2$ curbs memory use very effectively: the self-tracing program runs in constant space below 100 kb.

A Haskell Compilation: nfib-`nhc` Our second large-scale application is the `nhc` Haskell compiler itself. The compiler is a large program, consisting of about 100 modules with a total of over 14,000 source lines. When compiled for tracing, the compiler binary is 4.2 Mb in size, compared with 1.1 Mb for the non-traced binary. The compiler starts by reading, lexing, and parsing the Haskell input file. It then processes the interfaces for all imported files. The interface for the standard prelude is always processed. After that the source code (in the form of a syntax tree) is type checked and transformed in around 25 passes before finally the byte-code for the functions in the module is generated.

Traced programs are currently restricted to a limited repertoire of I/O operations. So for the purposes of tracing we modified the `nhc` compiler to take the source as standard input, and to deliver byte-code as standard output. Figure 13 shows the small source module supplied as input. Even compiling something as modest as the `NFib` module `nhc` takes 5.2 M reductions. But when the `Prelude` is trusted, 4.3 M of these are omitted from the final trace, leaving only about 900 k recorded reductions. We are able to trace the compilation without further pruning using around 70 Mb of memory to store the redex trail. The redex trail for the compilation can be accessed using the display program by clicking over individual instructions or operands in the byte-code output.

```
module NFib where

nfib :: Int -> Int
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + nfib (n-1) + nfib (n-2)
```

Fig. 13. Input for the nfib-`nhc` computation.

6 Related Work

The problem of tracing functional computations has been tackled previously by numerous researchers. In his recent thesis [11], Watson usefully reviews over twenty different schemes. But many proposed solutions have been unsuitable for full-scale applications. We concentrate here on a selection of previous work specifically addressing the problems of working with large computations.

Five years ago, Andrew Tolmach described in his thesis [10] a comprehensive tracing and debugging system for the *eager* functional language ML. A shorter and more recent account is given in [9]. Tolmach's system, like ours, worked by compile-time transformation of the program. He also happened to use machines very similar to our own (MIPS workstations with about 100 Mb of memory). He successfully applied his system to programs of up to 750 source lines, performing computations of over 50 million *events* (such as the binding of a value to a name — events are not directly comparable with reductions, but at a similar level). The overhead in execution time was impressively low: even compared with highly optimised normal compilation, the slow-down was only about a factor of three. However, the basis of tracing in Tolmach's system was navigation through a *sequence of states*, where state information was recorded only at check-point intervals of many thousand events — up to 256,000 for the largest application.

This interval, and the size of a *check-point cache* were the main ways of controlling memory use. Because of the sparse recording policy during traced execution, *repeated computation* was needed to provide information about events of interest to the programmer, whereas we record all available information in the trace structure. Though our aims are similar, at a detailed level our techniques are quite different, partly because of differences in evaluation strategy between an eager language such as ML and a lazy one such as Haskell.

In a paper at last year's IFL workshop [1] Jarvis and Morgan reported their experience extending the *cost-centre profiler* in Glasgow's GHC compiler [7] for use with large programs. Their motivating application was the LOLITA natural language system — at over 50,000 source lines one of the largest Haskell applications to date. Cost-centres are labels for parts of a program, added by the programmer as annotations (or by the compiler for default options such as one centre per top-level definition). Jarvis and Morgan introduced *cost-centre stacks* to gain precision when attributing costs of functions applied in many places, though the cost of maintaining distinct records for each permutation on the stack grows to tens of megabytes for LOLITA. As a side benefit, they found that the current stack of names provides a useful trace of the evaluation context when an error occurs in a large program. Full redex trails in our system carry all the information in a cost-centre stack, and far more besides.

Nilsson and Sparud use a trace structure similar in some respects to redex trails but with *forward* rather than backward links. To cope with larger computations they propose in [3] (and more fully in [2]) to build only some initial part of the trace when the program is run. When the programmer examines the trace, if a point is reached where the trace goes no further, the program is *re-executed* to obtain a trace extension at the point where it is needed. This process can be repeated many times. It is not clear how to use this incremental technique with backward traces, but the idea is interesting since it allows tracing in full detail for parts of a large computation.

We have already mentioned Watson's thesis [11] for its useful review. Watson's own approach to the tracing problem also involves transforming the program to augment values with traces of their computation, but he chooses to construct low-level traces recording the *sequence of evaluation*. His interpretive prototype incurs very high costs in both space and time, so he cannot demonstrate large applications. However, in a later chapter discussing the problems of scale, Watson discusses selective tracing and proposes a scheme much like our own for excluding trusted applications in a trusted context.

7 Summary, Conclusions and Future Work

We had two goals. The first was to extend our redex trail system so that in principle a *complete* record of a computation, including every reduction, can be made. In practice, complete records are often prohibitively expensive and overwhelm the user of the tracing system with too much information. So our second goal was to develop effective ways of specifying and constructing *partial* redex trails, recording only the reduction steps most likely to be of interest to the programmer. An important motivation for partial trails was to make tracing practical even for large computations.

We have attained both goals. Completeness has been achieved by extending our trail representation to incorporate indirections to exactly those applications previously unrecorded. By combining selective construction with controlled pruning we are able to trace much larger applications than before.

We conclude that the approach of compile-time transformation to generate redex trails is viable for a full-scale tracing system — something that implementations of lazy higher-order languages have lacked for too long.

We intend to develop the techniques further. As noted in Section 5.3, it would sometimes be more convenient to specify a bound on the size of heap memory than on the length of trails; this should be straightforward using a two-pass pruning routine. We also plan an option to archive redex trails to file as the computation proceeds: this will not only save space in main memory, but also allow traces to be re-examined as often as the programmer wishes.

The price we currently pay for these tracing techniques includes greatly increased execution times: programs take about *fifteen times* longer to run when constructing redex trails than when executed normally. This is not acceptable. We have begun work on low-level replacements for the traced application combinators defined in Haskell. Another help towards faster partial tracing would be some way to link modules compiled to build redex trails with others compiled normally. It is not clear how to achieve such linkage as the types of traced and untraced versions of a function differ; we suspect that naive conversion (stripping trails from values, or adding null trails to values) would be too expensive. By whatever means, however, our aim is to bring the speed of traced computation within a factor of two or three of normal speed in most cases.

We have said very little about the way programmers examine traces, though it is an important issue. We have constructed a special-purpose browser for redex trails, as described in [8]. This browser is normally applied when a computation is over, with the resulting output or run-time error as the starting point. But trail-building computations can also be interrupted, in which case the browser is attached to the trail for whatever is currently being evaluated; this mode of use allows non-terminating computations to be investigated.

Ultimately any claim to practical success in the field of tracing must be tested by experiments measuring the effectiveness of tracing tools in actual programming tasks. We hope to conduct such experiments in the near future. Meanwhile we have in our sights what many implementors would regard as the ultimate test of a tracer they themselves would wish to use — constructing an informative trace of a boot-strapping compilation.

Acknowledgements

Niklas Röjemo built the `nhc` compiler that we have adapted for our experiments in tracing. The work reported in this paper was funded by the Engineering and Physical Sciences Research Council (ROPA grant GR/K64334). Related implementation work was funded by Canon Research Europe.

References

1. S. A. Jarvis and R. G. Morgan. The results of: profiling large-scale lazy functional programs. In W. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop (IFL'96), Selected Papers*, pages 200–221. Springer LNCS 1268, September 1996.
2. Henrik Nilsson and Jan Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Research Report LiTH-IDA-R-96-23, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, August 1996.
3. Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Journal of Automated Software Engineering*, 4(2):152–205, April 1997.
4. H. Phillips. *The week-end problems book.* Nonesuch Press, 1932.
5. N. Röjemo. Highlights from nhc — a space-efficient Haskell compiler. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 282–292. ACM Press, June 1995.
6. N. Röjemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proc. Intl. Conf. on Functional Programming*, pages 34–41. ACM Press, June 1996.
7. P. M. Sansom and S. L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–85, March 1997.
8. J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, September 1997.
9. A. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.
10. A. P. Tolmach. *Debugging Standard ML.* PhD thesis, Princeton, USA, October 1992.
11. R. D. Watson. *Tracing Lazy Evaluation by Program Transformation.* PhD thesis, Southern Cross, Australia, October 1996.