J. Functional Programming 5 (2): 225–277, April 1995 © 1995 Cambridge University Press

Lexical profiling: theory and practice

CHRIS CLACK, STUART CLAYMAN AND DAVID PARROTT [†]

Department of Computer Science, University College London, Gower Street, London WC1E, UK
(email: {clack,sclayman,dparrott}@cs.ucl.ac.uk*)*

Abstract

This paper addresses the issue of analysing the run-time behaviour of lazy, higher-order functional programs. We examine the difference between the way that functional programmers and functional language implementors view program behaviour. Existing profiling techniques are discussed and a new technique is proposed which produces results that are straightforward for *programmers* to assimilate. The new technique, which we call *lexical profiling*, collects information about the run-time behaviour of functional programs, and reports the results with respect to the original source code rather than simply listing the actions performed at run-time. Lexical profiling complements implementation-specific profiling and is important because it provides a view of program activity which is largely independent of the underlying evaluation mechanism. Using the lexical profiler, programmers may easily relate results back to the source program. We give a full implementation of the lexical profiling technique for a sequential, interpretive graph reduction engine, and extensions for compiled and parallel graph reduction are discussed.

Capsule Review

This paper describes a profiler for lazy higher-order programs evaluated by graph reduction. Application programmers are the intended users of the profiler. The costs of evaluating expressions are assigned to lexical units enclosing them in the source program: the effect is much *as if* programs were executed using a 'call by value' rule; yet it reflects savings due to actual 'call by need' evaluation. The prototype implementation is hosted by an interpreter, and the paper discusses its application to some small examples with promising results.

To ensure accurate results in the presence of sharing, all components with shared uses have to be treated as independent units for profiling purposes. Also, as the process of graph reduction is quite heavily embellished, self-reference is an issue for the execution-time profiler (see Section 5.6). It will be interesting to see how effective the scheme is for full-scale applications — but that must await a future implementation in a widely distributed compiler.

1 Introduction to profiling

The aim of program profiling is to enable the programmer to use the resulting information to determine whether parts of the program consume a disproportionate amount of resources. For many real-world applications it is not just desirable but

[†] During the course of this work D. J. Parrott was supported by an SERC studentship.

essential for a programmer to be able to monitor and subsequently alter the time and space behaviour of the program. Without profiling information, it may be impossible to rectify a program which exhibits degenerate behaviour.

Lazy, higher-order functional languages provide a programming framework which is far removed from the details of instructing computer hardware. This high-level framework enables a programmer to express problem solutions in a way that closely resembles the problem specifications and which may exploit new software-engineering techniques (Hughes, 1989). Unfortunately, the high level of abstraction means that the executable form of a functional program is often unrepresentative of the original source code, and *vice versa*. The source code may be amenable to a limited amount of complexity analysis, though this is not able to take into account the algorithms used and the associated overhead incurred by the underlying evaluation mechanism. Programmers rarely have intimate knowledge of the evaluation mechanism and it is therefore difficult for them to reason about the time and space complexity of a functional program. This contrasts strongly with imperative languages, where the source code is a key factor in estimating a program's behaviour prior to execution.

If programmers find it difficult to anticipate run-time behaviour, they are likely to find it even more difficult to interpret run-time profiles in order to reason about which sections of a program need to be modified. A profiler is only worthwhile if it produces information that can help a programmer to reason about improving programs.

To improve the quality of measurement tools for functional languages, we have designed a profiler primarily for use by application programmers rather than functional language implementors. Our profiler provides information that is related to the way the program is written rather than to how it is evaluated; thus enabling programmers to easily relate results back to the source program. The results directly reflect the *lexical* scoping of the source program, thus overcoming problems caused by compile-time program transformation, lazy evaluation and higher order functions. We call the technique *lexical profiling*.

The profiler counts function calls and accurately profiles the time and heap space used by lazy, higher-order functional programs. It can be used to monitor programs as they run and to build detailed trace information for post-mortem analysis and debugging.

In the remainder of this section we discuss what the user expects from a profiler and the different possible profiling techniques. In the next section we present a number of existing profilers and discuss their merits. We then proceed to describe our new profiling technique and give full details of compilation and run-time implementation. We conclude the paper by providing a few short examples of the profiler in use and discussing directions for further work.

1.1 Implementor vs. programmer views of profiling

A fundamental difference exists between the way in which application programmers and system implementors naturally reason about the execution costs of functional languages which use call-by-name evaluation semantics. In a call-by-value language the source-level definition of expressions corresponds closely to the order of their evaluation at run-time. For example, the run-time behaviour of:

$$x = f_{cbv}\,(exp_1, \ldots, exp_n)$$

is described as follows:

1. All of the arguments $exp_1, \ldots, exp_n$ are evaluated (this is independent of the definition of f_{cbv}).
2. The function, f_{cbv}, is executed. Any subexpressions contained within f_{cbv} and relevant to the result are evaluated.
3. The result is bound to x.

In a call-by-name language, delayed evaluation has the effect of transferring the evaluation of an expression from the position in the code where it is declared to the point where its value is required. Consider the following example:

$$y = f_{cbn}\,(exp_1, \ldots, exp_n).$$

Without a definition for f_{cbn}, and further information about the context in which y is used, the run-time behaviour cannot be determined beyond stating that if y is evaluated then the function will be called and the result bound to y. Any combination of the arguments may be evaluated or may remain unevaluated by the function call. Moreover, the result may contain references to some unevaluated arguments which may then be evaluated at a later time. Consider the following definition of f_{cbn}:

$$f_{cbn}\,(arg_1, \ldots, arg_n) = (arg_1, arg_n)$$

Here, the result is a *pair* containing the first and last arguments passed to f_{cbn}. The pair can be constructed without evaluating *any* of the arguments. If the result, y, later occurs in an expression of the form:

$$z = (fst\;\; y) + (snd\;\; y)$$

then the values of exp_1 and exp_n will finally be needed and the expressions will be evaluated. This is a simple example which clearly illustrates how difficult it can be to reason about *where* expressions are evaluated. In large functional programs the examples become much more complex, and unevaluated objects may be passed through many function calls before evaluation occurs.

The difference between the call-by-value and call-by-name cases is significant. For straightforward performance evaluation, application programmers find it simpler to reason about the behaviour of call-by-value programs than of call-by-name programs because expressions are always evaluated by the functions in whose bodies they occur (see, for example, Nilsson and Fritzson's work on algorithmic debugging (Nilsson and Fritzson, 1992)). To simplify reasoning about call-by-name functional programs, we propose that the cost of evaluating expressions should be reported to application programmers with respect to the lexical structure of the source program. The program is still evaluated using call-by-name semantics but the call-trace is constructed *as if* call-by-value evaluation had occurred. For example, in the following definition:

$$f\;g\;x = g\;expensive\;x$$

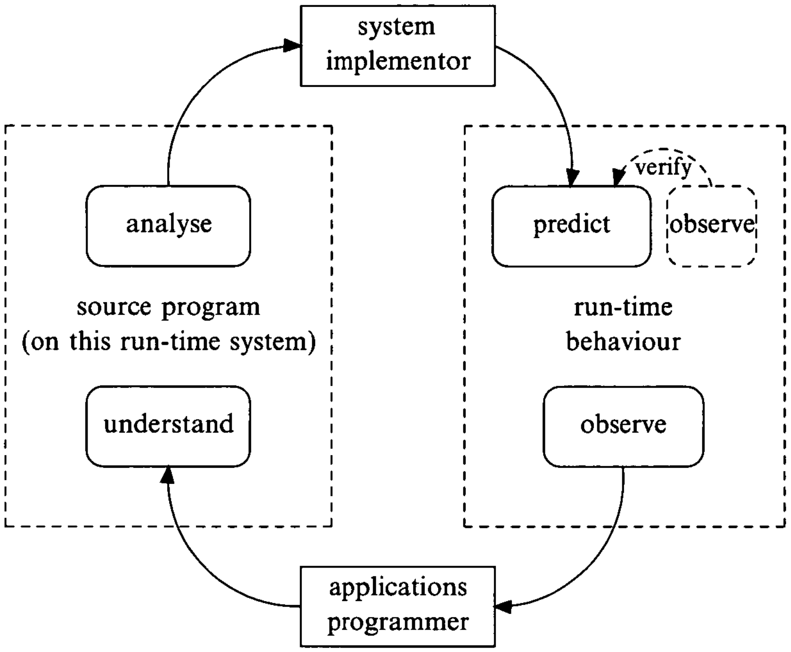

Fig. 1. How applications programmers and systems programmers relate functional programs to their run-time behaviour.

(where g is non-strict in its first argument), call-by-name evaluation will cause *expensive* to be evaluated in or below g (if it is evaluated at all); but call-by-value evaluation would have caused *expensive* to be evaluated in f, prior to calling g. We continue to treat *expensive* as a call-by-name argument, but if it is evaluated its execution time is attributed to f. If the *expensive* expression is exhibiting degenerate behaviour, the application programmer's attention is then drawn immediately to the body of function f.

A system implementor takes a very different view of the situation. The actual behaviour of the program at run-time is more important than the lexical relationships within the original source code. The effects of call-by-name evaluation must be reported *exactly as they occur* so that the knowledge can be used to improve compile-time and run-time heuristics (e.g. dynamic scheduling).

Despite having different requirements and viewpoints, both application programmers and system implementors need to make a connection between the definition of call-by-name functional source code and its eventual run-time behaviour. This is illustrated in Fig. 1 where:

The application programmer observes the run-time behaviour of the program and attempts to map this *back* to the original definitions in the source code.

The system implementor analyses the source program and attempts to use the analysis to *predict* its run-time behaviour. The run-time behaviour is also observed by the implementor, in order to verify the success of the prediction. Analysis of the run-time behaviour is made with respect to the run-time domain, hence it is not necessary to map the run-time behaviour back to the lexical structure of the source code.

In the first case the programmer performs a backward mapping from run-time behaviour to the source program, and in the second case performs a forward mapping from the source program to run-time behaviour. The difference between the two viewpoints has a large effect on the way that profiling and monitoring tools are constructed.

Sharing

The above discussion has focused on the different views taken by programmers and implementors. We have proposed that in order to provide useful information, a profiler should attribute costs as though call-by-value evaluation had taken place instead of call-by-name evaluation. However, a lazy functional language also supports optimized evaluation of shared subexpressions.

In a lazy functional language a shared subexpression is evaluated at most once, by whichever part of the program first requests its evaluation. Sharing is often used to make programs more concise, but it is also an important tool used by application programmers for optimizing resource usage. It is therefore essential that a profiler should reflect the changes in resource consumption which are related to the use of sharing.

Call-by-name evaluation can also be used to optimize the performance of a program, and indeed the lexical profiler proposed in section 3 faithfully reports performance gains due to call-by-name evaluation. However, the lexical profiler attributes these costs to program functions as though call-by-name evaluation had occurred. By contrast, the lexical profiler makes sharing apparent to the programmer rather than pretending that sharing had not taken place. We take the view that the computational model for sharing is far simpler than that for call-by-name evaluation (which results in an interleaving of evaluation which is difficult for an application programmer to predict) and that it is important for a profiler to present information about sharing.

Sharing is a global property of a program: to discover whether a top-level function is shared it is necessary to inspect the whole program. Thus, in order for a profiler to provide useful information regarding sharing it must *profile the entire program at once*. We will return to the issue of sharing in sections 3 and 4.1.

1.2 Different kinds of profiling

There are at least three different kinds of profiling that can be undertaken in functional programming environments:

Program profiling: Measurements relate to the program's behaviour. The profiler may allow the programmer either to measure the one-off cost of an individual expression inside a function body, or to measure the cost of an individual function as an accumulation of the costs of all the applications of that function (of course, multiple expressions or multiple functions may be measured at the same time). Furthermore, the profiler may determine the costs either *dynamically* or *lexically*: in the former case, costs are attributed to the function which is the immediate cause of the resource utilization, whereas in the latter case costs are attributed to the function whose body contains that expression which defines the resource utilization. The difference between dynamic and lexical profiling is important for systems which use lazy evaluation.

Abstract machine profiling: This measures the effectiveness of the run-time system by examining the overheads of function calls, function returns, heap management, garbage collection, etc. (for example, see King, 1990).

Task profiling: This is particularly relevant in parallel environments where programs are divided into tasks which execute on separate machines. The number and size of the tasks are reported (see our related work in Parrott and Clack, 1992; Parrott, 1993).

The last two kinds of profiling are clearly of more interest to functional language *implementors* rather than functional *programmers* and will not be considered further. The first kind of profiling is directly relevant to programmers, but the distinction between the dynamic and lexical styles is subtle. With a strict language, the two styles are identical, but in a lazy language the difference between the two styles is more fundamental. Lexical profiling measures *whether* work happens, and how much happens, with results being presented with respect to the source code. By contrast, dynamic profiling measures *when* work is done.

Figure 2 presents three example programs which may be used to illustrate the difference between dynamic and lexical profiling. These three programs perform the same set of arithmetic operations but differ in where the expression x is declared and evaluated (assuming lazy evaluation throughout). Table 1 shows the number of primitive operations counted for the functions in each program using both lexical profiling and dynamic profiling. The results of the lexical profiler always show the cost of x being associated with the function in which x is lexically contained. The results of the dynamic profiler highlight the presence and effect of laziness, and the cost of x is associated with the function that required the value of x.

Although we may not be interested in precise numbers of primitive operations, these examples illustrate the differences in the two profiling styles and show that in order to appreciate fully how a program is evaluating, *both* styles may be used together to provide a comprehensive view. If we consider just the count of primitive operations using the lexical profiling style, the results given in Table 1 for programs 1 and 3 are indistinguishable. Similarly, if we count just the primitive operations reported dynamically then programs 1 and 2 are indistinguishable. Functional language implementors can therefore benefit from using a lexical profiler in conjunction with a dynamic profiler. Functional programmers, however, would require detailed knowledge of the underlying evaluation mechanisms to use a dynamic profiler effectively.

Let us now consider an example in which dynamic profiling may give different results each time the program is executed but where lexical profiling will give consistent results. In the program:

Table 1. *The difference between dynamic and lexical profiling.*

Program	function in which x is...		Number of primitive operations					
			lexical profile			dynamic profile		
	...declared	...reduced	f	g	h	f	g	h
1	f	h	$1 + p_x$	1	1	1	1	$1 + p_x$
2	g	h	1	$1 + p_x$	1	1	1	$1 + p_x$
3	f	g	$1 + p_x$	1	1	1	$1 + p_x$	1

p_x = number of primitive operations to evaluate x

Program 1
```
f    =  (g x) / 18
        where x = ⟨expression⟩
g x  =  (h x) * 10
h x  =  x + 12
```

Program 2
```
f    =  (g 20) / 28
g y  =  (h x) * y
        where x = ⟨expression⟩
h x  =  x + 22
```

Program 3
```
f    =  (g x) / 38
        where x = ⟨expression⟩
g x  =  x * (h 30)
h x  =  x + 32
```

Fig. 2. The three programs whose behaviour is analysed in the above table.

```
f   = (g x) + (h x)
      where x = expression
g x = h x * 10
h x = x + 32
```

the evaluation order of the primitive + is important. If the order is left to right then a dynamic profiler will credit g with the evaluation of x, but if the evaluation order is right to left then h will be credited. In a parallel system where the load balance and evaluation order are non-deterministic, a dynamic profiler may return different results on different occasions. Lexical profilers do not suffer from either of these problems as results are associated with lexical scope, thereby providing a static relationship between the source code and the run-time results.

A corollary of this relationship is that the interpretation of lexical profiling results is independent of the underlying evaluation mechanism (though, of course the

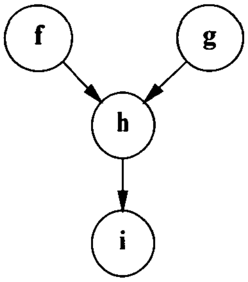

Fig. 3. An example call graph.

same program might exhibit differing performance on differing implementations). In contrast, the data from dynamic profiling is always dependent on the evaluation mechanism.

1.3 Methods of propagating profile-times

When a function is profiled, it is often desirable for the execution time reported for that function to include the times of its subfunctions ('children') (Graham *et al.*, 1982). In this section we discuss two complementary methods for propagating the costs of functions to their parents. We call these *statistical* and *inheritance* profiling, respectively. The lexical profiler described in this paper is amenable to both of these methods of propagation.

For propagation to be 100% accurate, a profiler would need to reconstruct the entire call-path for all function calls. However, the time and space costs associated with building and analyzing complete program traces are prohibitive. In practice, therefore, it is more common to log information concerning just the calls made by each function to its immediate children (Graham *et al.*, 1982). This is obviously less comprehensive than a full program trace, and consequently leads to approximations when profiled attributes are propagated from child functions to their grandparents. For example, consider the following code:

```
f a = h a + 1
g b = h b - 1
h x = i x + i x
i x = x + 1
```

Figure 3 shows the call graph for these function definitions. It is clear from the call graph that the function `i` is called only from function `h`, but that `h` is called from both `f` and g.

When the code sequence is profiled we log separately the time spent in `h` due to calls from `f` and the time spent in `h` due to calls from g. We also log the time spent in `i` due to calls from `h`. The information recorded about the calls to immediate children is therefore accurate. A problem arises, however, when we come to calculate a *total* time for function `f` (i.e. *including* the time spent in its descendant functions `h` and `i`). Without storing a complete call-path from `f` to `i` we are uncertain how much of the time spent in `i` was ultimately due to calls originating at `f` and how much was due to calls originating at g.

1.3.1 Statistical propagation

One solution to the above problem (Graham *et al.*, 1982) is to divide `i`'s time according to the ratio:

$$\frac{\text{number of calls from } \mathtt{f} \text{ to } \mathtt{h}}{\text{number of calls from } \mathtt{g} \text{ to } \mathtt{h}}$$

(e.g. if there are six calls from `f` to `h` and 4 calls from `g` to `h`, then `f` will get 60% of the time in `i` and `g` will get 40% of the time in `i`). This is the method used by *statistical* profilers such as `gprof`: it assumes a linear correlation between the number of calls to one function and the execution time of another. For the example code given above the assumption is correct. However, in more complex examples where calls to `i` are dependent on parameters passed to `h`, the correlation is less likely to hold and the results may therefore be inaccurate.

1.3.2 Inheritance propagation

We observe that the process of propagating the time-costs of a subgraph to the root node of that subgraph is equivalent to collapsing the subgraph to a single point. The potential inaccuracy of the statistical style of propagation is due to incomplete information being available when a subgraph is collapsed. Often, this problem may be avoided by collapsing a subgraph prior to measuring costs; of course, it is the profilers *view* of the graph that is collapsed, and not the actual call-graph.

For example, in Fig. 3 the subgraph whose root is `h` may be viewed as a single node. Thus, for profiling purposes the sub-function `i` is treated as an extension of its parent, and the total amount of time spent both in and below `h` may be propagated accurately to `f` and `g`. The code outline would be profiled as though it had been written thus:

```
f a  = h a + 1
g b  = h b - 1
h x  = i x + i x
       where
       i x = x + 1
```

The above example demonstrates the use of inheritance to provide more accurate information in the presence of sharing. However, we can also apply the inheritance mechanism to *arbitrary* subgraphs to allow the programmer control over the amount of profiling information that is reported. As an aid to further discussion, we define the following terms (we use the term 'function' to refer to either a function or a CAF:[†] this is discussed further in section 5.5):

An unprofiled function is a function which is subsumed into a parent during the inheritance process – *unless it is a shared function*. Shared, unprofiled functions cannot be inherited because there is an ambiguity as to which parent should subsume the child. This issue is discussed further in section 4.

[†] Constant Applicative Form (Peyton Jones, 1987b).

A profiled function is a function which is *not* subsumed into a parent during the inheritance process.

A profiler that implements inheritance propagation can also provide the equivalent of statistical propagation; all that is necessary is to post-process the results. However, mutual recursion complicates the process and we leave the provision of a statistical-mode post-processor to further work.

2 Existing profilers – a brief history

In this section we describe existing profilers for both imperative and functional languages and consider how they have motivated and affected the design decisions for our profiler.

2.1 gprof – *an existing imperative profiler*

The UNIX profiling tool **gprof** (Graham *et al.*, 1982) produces a profile of a program based on the program's dynamic call tree. Results are presented with an entry for each profiled function, together with information about the parents and children of the profiled functions. The data for child functions is propagated towards the root of the call tree so that each node incorporates a measure of the expense of its subtrees. The **gprof** profiler is based on an older program, **prof**, which does not propagate information in this way but just reports how many times each function is called, the amount of time spent in each function, and the percentage of total running time for each function. The **gprof** mechanism is a great improvement over the simpler *flat* style of profiling and, as a result, **gprof** has been used successfully with imperative programs for many years.

The implementation of **gprof** is based on the assumption that code is statically placed in consecutive memory locations at load time. The execution time of each function is not measured exactly, but approximated by monitoring the location of the program counter every 1/60th of a second. A histogram of program counter values is constructed and the amount of time spent in each function is estimated by post-processing the histogram in conjunction with a map of code locations. One problem with **gprof** is that it does not monitor space utilization and so cannot provide full information for programs which make extensive use of dynamic memory allocation (however, the **mprof** profiler (Zorn and Hilfinger, 1988) does provide this facility). Finally, we note that **gprof** does not provide useful information for mutually recursive functions because it collapses each strongly-connected component in the call graph to a single point.

Despite the faults and inaccuracies mentioned above, **gprof** has proved to be a useful tool for imperative programmers. This provides a motivation to develop similar profiling tools for functional languages.

Table 2. *Comparing* string_to_int *functions.*

Input	string_to_int1		string_to_int2	
	cells	reductions	cells	reductions
""	10	3	19	7
"1"	19	9	47	23
"12"	29	14	77	41
"123"	39	19	107	59
"1234"	49	24	137	77
"12345"	59	29	167	95
"123456"	69	34	197	113

2.2 Statistics provided by functional language implementations

Many functional language implementations (e.g. Miranda[‡] (Turner, 1985)) provide rudimentary statistical information detailing such things as the number of reductions performed, and the total number of heap cells allocated during an evaluation. These can be used to choose between two different program designs. For example, consider the following two functions which both convert a string to an integer:

```
string_to_int1 :: String -> Int
string_to_int1 s
    = string_to_int' s 0
      where
      string_to_int' []    v = v
      string_to_int' (h:t) v
          = string_to_int' t (10*v + (ord h - zero))
      zero  = ord '0'

string_to_int2 :: String -> Int
string_to_int2 s
    = sum [ x*y | (x,y) <-  scale_factors ]
      where
      scale_factors = zip (reverse digits) (iterate (*10) 1)
      digits = map ((\v -> v - ord '0').ord) s
```

If we wish to determine which is the most efficient, we can compare their run-time behaviour for a given input. Table 2 gives the number of cells allocated and the number of reductions performed for the given input. (This experiment was conducted using the Haskell interpreter, gofer.)

This data shows that although both functions display linear behaviour, one function is more efficient than the other and we would probably choose string_to_int1 to convert strings to integers in an application. Thus, statistical information from the run-time system can sometimes be very useful; however, the information is limited and if more information is required then a program profiler must be used.

‡ Miranda is a trademark of Research Software Ltd.

2.3 The New Jersey SML profiler

The New Jersey implementation of Standard ML is remarkable for the fact that it is supplied with a functional language profiler (Appel *et al.*, 1988) which gives more extensive information than the simple statistics discussed in section 2.2. The programmer chooses a subset of functions to profile and, for the purposes of profiling, the children of these functions are subsumed into their parents. This is *inheritance profiling* (see section 1.3). The accompanying manual directs the programmer to experiment with multiple profiles of the same program, choosing different groups of functions each time in an attempt to get a more accurate idea of the program's behaviour.

The profiler is designed to profile strict evaluation and does not profile usage of heap space. Also, the results of the SML profiler can be difficult to interpret in the presence of higher-order functions because the execution times of higher-order arguments are attributed to special identifiers instead of to the real functions. The authors of (Tolmach and Dingle, 1990) argue (with an example) that the ambiguous results are of little consequence in short programs where a higher-order function is called just once; they then suggest that the programmer should be able to *guess* to which real function the special name refers. However, guessing is not so simple for large programs where higher-order functions (such as `map`) are called repeatedly with different higher-order arguments each time. The SML profiler coalesces all applications of a single higher-order parameter into a single timing, thus losing vital information (Appel *et al.*, 1988).

2.4 UCL in-line cost primitive

An early profiling technique investigated at UCL was the use of in-line cost functions (Parrott and Clayman, 1990) for measuring the cost of evaluating an expression. The technique used a cost function which had the equivalent semantic behaviour to the identity function. In order to ensure referential transparency, the cost of the evaluation was written to a special output stream which could not be accessed by the program. For the primitive to work properly, we had to ensure that evaluation took place at the right time for measurements to be meaningful. Normally, an application of the identity function would not cause its argument to be evaluated. However, we desired the argument of the cost function to be evaluated to the extent demanded by the *context* in which the application occured. For example, in the expression `n + fst (cost expr)`, `expr` is a pair whose first element is a number. In this context, the cost to be measured is that of constructing the pair and evaluating the first element. To ensure that the correct amount of evaluation was measured by the `cost` primitive we therefore advocated the use of a run-time system which incorporated Burn's evaluation transformers (Burn, 1987) to provide the necessary run-time information.

However, the in-line cost functions have drawbacks because the information provided by a cost function is dependent upon its context at run-time. It is impossible to interpret the results without thoroughly understanding the effects of laziness on the evaluation of a program. In a *parallel* implementation, the order in which expressions are evaluated cannot be determined and the timings returned by `cost` will change from one program run to another. A fundamental problem with this profiling technique is that it takes a microscopic view of the program – we prefer a macroscopic view which reports its results at a level of abstraction understood by the functional programmer. Furthermore, the cost functions do not provide information about space usage or function call-counts.

2.5 Glasgow cost centres

Sansom and Peyton Jones (1992) introduced *cost centres*, which associate unique names with the time and space costs of evaluating specified expressions. The implementation is an extension of the Spineless Tagless G-Machine (Peyton Jones and Salkild, 1989) used by the Glasgow Haskell compiler. When an expression is to be profiled, the closure built to represent the expression is augmented with an identifying tag which remains with the closure throughout its life. A register is added to the STG-Machine in order to keep track of the *current* cost-centre. The register is sampled at intervals to determine the amount of time spent in each profiled expression. The key to the implementation of the Glasgow profiler is to ensure that the current cost-centre register is set at appropriate times. The issues are complex and are discussed in detail in Sansom's PhD thesis (in preparation). We describe a similar mapping of our own profiling scheme to the TIM compiled abstract machine in (Clack and Parrott, 1993).

Cost centres are the subject of current research and during the last year the technique has been subject to many changes (see, for example, Sansom, 1993, 1994). It is encouraging to learn that cost centres have now adopted the lexical approach advocated in this paper. As a result, cost centres exhibit similar properties to those described in section 3 of this paper. There are, however, two basic differences between cost centres and the profiling scheme described here:

1. Cost centres are used to profile expressions: we choose to profile functions. This distinction is largely stylistic, except for the treatment of shared expressions. Cost centres require all Constant Applicative Forms (CAFs) to be profiled separately, even if the programmer has not asked for these to be profiled, but does not require shared functions to be profiled separately. By contrast, the profiling scheme described in this paper does not distinguish between functions and CAFs and only requires a CAF or function to be profiled separately if it is shared. In section 5.1.3 we explore the consequences of different methods of specifying which entities are to be profiled.
2. The implementation of cost centres requires a smaller space-overhead than the implementation of our scheme. However, we believe that this is at the expense of requiring complex transformations and the special treatment of CAFs. We return to this issue in section 5.1.5.
3. The method of observing time is different: both Glasgow's method and our method are subject to inaccuracies, but in different ways.

2.6 Runciman and Wakeling heap profiler

Runciman and Wakeling (1993) describe a profiler that monitors heap usage of lazy, functional programs. Each cell allocated in the heap is tagged with a 'producer' name (the name of the function which created the cell) and a 'construction' name (the type of construction which the cell represents; typically the name of a constructor of an algebraic data type). The memory occupied by the live heap (in bytes) is discretely sampled and plotted as a function of time (in seconds); the area covered by this graph is reported as a single (time × space) product at the top of the display and is used as a measure of the cost of the program. One can also request heap-profiles which are restricted to named functions (either as producers or consumers) – such selection helps to 'home in' on a space fault. A great deal of attention has been given to the presentation of the profiling results, to good effect.

A worked example in Runciman and Wakeling (1993) shows that heap profiling can be very useful to functional language implementors, since the authors were able to detect two serious faults in the abstract machine. Of course, an applications programmer would not normally have access to the source code of the underlying evaluation mechanism, and probably would not have the intimate knowledge of the system which would enable her to detect and fix such faults. However, the worked example in Runciman and Wakeling (1993) also shows how heap profiling can help applications programmers to detect pipeline blockages by profiling **producers**, and to detect programming blunders and algorithmic inefficiencies by profiling **constructors**.

The Runciman and Wakeling profiler is undoubtedly a major step forward, but it is deficient in a number of areas:

1. The producer profile can indicate that a certain function (e.g. `map`) is responsible for producing a disproportionate amount of cells; but there may be many applications of the function in a program and the producer profile cannot distinguish between the different applications.
2. There is (deliberately) no information about the time spent in functions or the number of times each function is called.
3. Heap space is measured by visiting the whole graph at pre-determined intervals. For large heaps (as in Runciman and Wakeling's example), the pauses caused by these visits will be long. Thus, for practical reasons, an upper bound is imposed on the sample frequency and this can cause the presented data to be misleading due to quantization (though experience of heap profiling sequential implementations (Runciman and Wakeling, 1992; Kozato and Otto, 1993; Grant *et al.*, 1993) suggests this is not problematic). A more acute problem for distributed-memory parallel implementations is that this method will incur high overheads, since in such systems a global graph walk can be extremely expensive.

3 Lexical profiling

In this section we present a new technique for profiling lazy, higher-order functional programs. We call this technique *lexical profiling*.

We aim to provide applications programmers with a straightforward method of attributing the run-time resource consumption of their programs to the relevant function definitions within the source code. Lexical profiling attributes the costs of all, and only, those expressions *textually* contained within each profiled function to that function. The advantage of lexical profiling is that it provides information that is related to the way the program is written rather than to the way it is evaluated. Consider the following short program:

```
let map f []      = []
    map f (x:xs) = f x : map f xs
    increment x  = x + 1
    g = map increment [1..1000000]
    h = map increment [1..10]
in (g, h)
```

When this program is executed, `increment` will be invoked many times from within `map` via `map`'s higher-order parameter. The explicit references to `increment`, however, are *lexically* enclosed within the definitions of g and h. From the programmer's point of view it makes sense to talk about the function `increment` which *originates* from either g or h in preference to that which was *invoked* by `map` because this is more readily related back to the source code. This is the essence of lexical profiling. In larger examples where `increment` is passed as a higher-order argument to many functions, the information provided by the lexical profiling style is far simpler for the programmer to assimilate than the non-lexical alternative.

To achieve lexical profiling, we must solve the following two problems:

1. Higher order programming languages allow functions to be passed as arguments to other functions and to be returned as results. Consequently, a function application may be denoted in one part of a program and evaluated elsewhere. Thus, the attribution of costs is non-trivial.
2. Lazy evaluation requires that objects are evaluated only as far as is necessary for immediate use. Consequently, evaluation is temporally distributed and the evaluation of several expressions can be interleaved. Correctly attributing cost which is interleaved with other evaluation is similarly non-trivial.

In the remainder of this section, we provide further discussion of the technique and the problems of higher-order and interleaved computation. Subsequent sections explain how we solve the above two problems.

3.1 Properties of the lexical profiler

The lexical profiler collects statistics for user defined functions for either *all* top-level functions or just those which the programmer requests.[§] The profiler measures the time and space used at run-time by profiled functions and reports the number of calls made to profiled functions (and from where they originated). For lexical

[§] Requests are expressed using compiler options rather than in-line program annotations.

profiling the profiler must recognize when lazy arguments are being evaluated and switch context so that the time and space required for the evaluation are attributed to the function whose definition lexically contains the associated expression. The context switch does not constitute a full function call so the number of calls made must not be incremented.

Lexical profiling requires the compiler to record the lexical scope of functions so that the run-time system can monitor the functions and attribute measurements correctly in the presence of higher-order functions and lazy evaluation. The compiler needs to access the source program early in compilation and is responsible for maintaining the lexical affinities throughout all subsequent program transformations.

The run-time system is responsible for measuring the time spent in a function, the number of calls to a function, and the amount of space used by a function. The simplest of these is the number of **calls** to a function. This represents the number of times that the function is applied to its full quota of arguments.

The **space** used by a function is defined as the number of heap cells which have been allocated by that function and which have not yet been garbage collected. This heap occupation is a function of time, whose graph is plotted (it may be helpful to implementors if we distinguish those cells which are purely used for system purposes, such as cells used to hold suspended tasks). We define space usage in terms of the garbage collector activity because it is more useful for a programmer to know exactly how much memory is actually being used, and is therefore not available for re-use, than to be given information about how much memory constitutes the accessible program graph (which is an implementation issue). Knowledge of actual memory use will, for example, show whether the program will soon exhaust the available heap memory.

If space usage is defined in terms of those cells that are not available for re-use then the continuously-measured space profile generated by our profiler will be characteristically different according to the type of garbage collector being used. Figure 4 illustrates the difference between the characteristic space profiles for a reference-counting garbage collector and for a mark-and-sweep or two-space copying collector. Reference-counting collectors (such as the one used by our graph reduction engine) are typically more distributed in time than the other kinds of collectors and the function of heap usage against time more closely follows the implementor's view of the accessible heap. By contrast, the use of mark-and-sweep collection means that the measured function is effectively a quantized version of the accessible heap: the number of accessible cells in the program graph only equals the number of cells in use immediately after a sweep operation and between sweeps only cell allocations (not deallocations) are measured. Reference-counting does not, however, provide a perfect match between accessible cells and those in use: cyclic structures might not be collected at all, in which case the cells in use might be greater than the accessible cells (as shown in Fig. 4).¶

¶ Cyclic cells can be collected by reference-counting garbage collectors: furthermore, the collection of cycles can be distributed in time (e.g. see Wright, 1994).

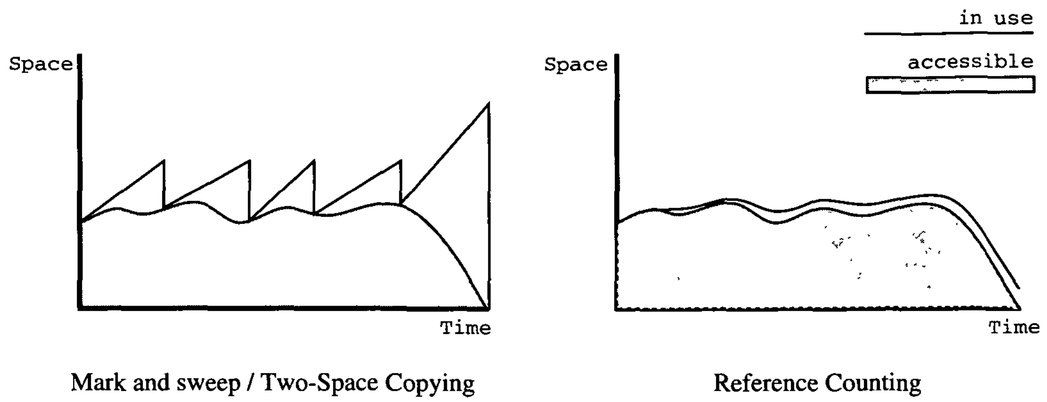

Fig. 4. Space usage with different garbage collectors.

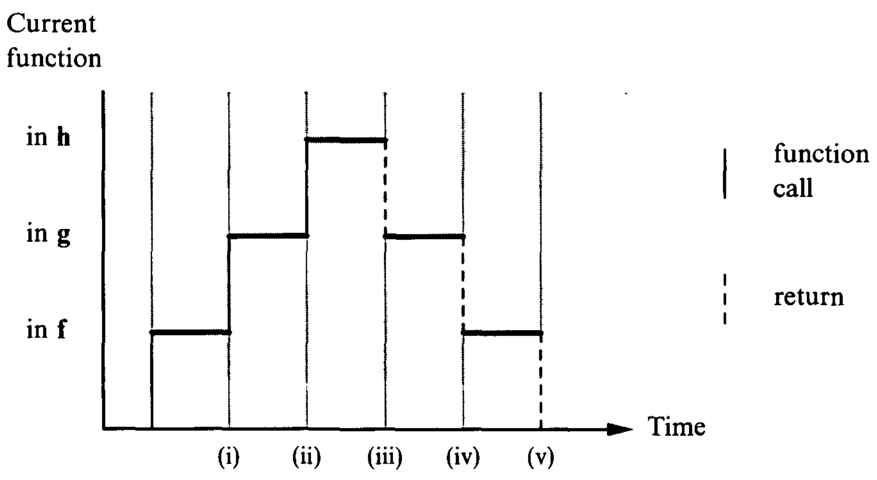

Fig. 5. Recording processing time under strict evaluation.

From the programmer's point of view it does not matter which garbage collection technique is used, since the profiler will faithfully reflect the actual space utilisation of the program. We use reference-counting garbage collection in our implementation.

The **time** spent in a function is the accumulation of small amounts of time evaluating different parts of the function. This is illustrated in Fig. 5 (based on the simple program presented in section 1.2), where time increments are recorded at points (i)–(v) during the evaluation (i.e. when a function is called or when a return is made). We do *not* include garbage collection time, which is recorded separately.

3.2 Profiling in the presence of lazy evaluation

Under lazy evaluation, an expression appearing lexically within the function body of one function might be evaluated by another function. For lexical profiling, the time and space resources consumed during the evaluation of the expression should be attributed to the function in which the expression was denoted, not that in which it's evaluation was forced (the lexical attribution makes the programmer's job simpler when searching for expensive code).

The following function definition illustrates the point:

$$\texttt{f = g exp}_A\ \texttt{exp}_B$$

Here, the expressions $\texttt{exp}_A$ and $\texttt{exp}_B$ originate from the definition of function `f`. However, lazy evaluation semantics demand that each expression is only evaluated when its normal form is definitely required; in this case neither expression will be evaluated by `f` because they are merely passed as arguments to g. The lexical profiler must therefore keep track of the origin of *all* argument expressions so that when the expressions are eventually evaluated the profiler can assign the cost of the evaluation to the correct function.

To correctly (lexically) attribute time costs, the profiler must therefore be able to make a context switch. Figure 6 (once again based on the simple program in section 1.2) illustrates a context switch for a function `f` which passes an argument lazily to a function g. When the evaluation of the argument is forced within the body of g (at point (ii) in the figure) the profiler must record the time required to evaluate the argument separately, and attribute it to `f`. When the evaluation of the argument is complete (point (iii) in the figure), the profiler switches context again, and continues to attribute time to g.

To log the time increments, a *profile table* is built for every profiled function. For each profiled function f_p, the table contains an entry which records the functions from which f_p originates and, for each unique origin, the number of calls to f_p and the accumulated processing time due to those calls. Each *profile table* is a representation of calls from multiple parents to a single child. To generate the full call graph the data for a single parent to multiple children is needed. This data can be generated by inverting the *profile tables*. When the call graph is completed, the profiling results are evaluated and returned to the user.

4 Compiling with profiling in mind

The profiler presented in this paper is amenable to both fully compiled and interpretive abstract machines. To simplify the presentation we demonstrate the general principles of the profiler's implementation using a sequential, interpreted model of graph reduction. In this section we discuss modifications that must be made to compilers. In section 5 we shall present details of the run-time mechanisms which facilitate call-count profiling, time profiling, and space profiling, and in section 9 we discuss how these techniques extend to fully compiled abstract machines.

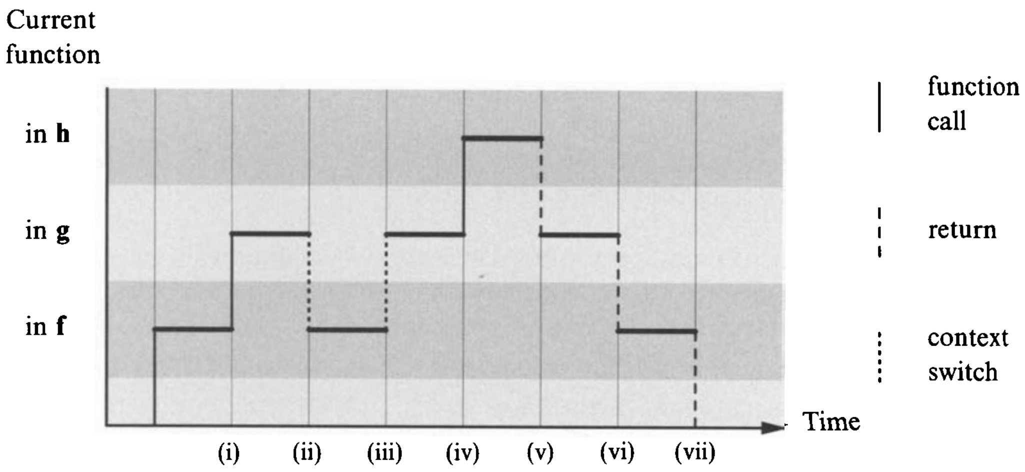

Fig. 6. Recording processing times in the presence of non-strictness.

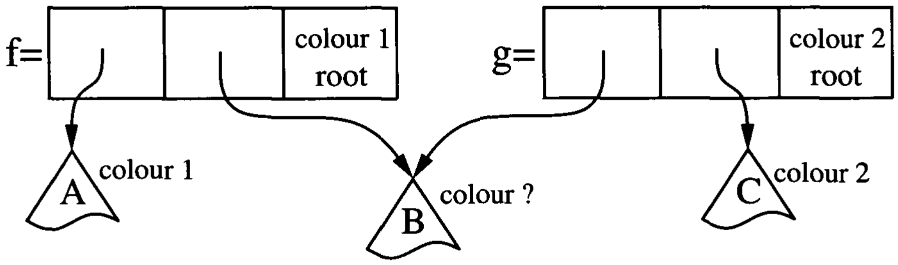

Fig. 7. An example of an unprofiled function which is shared by two profiled functions.

The following discussion assumes that functional programs are represented internally by the compiler as a parse-graph (e.g. see Parrott and Clack, 1991) and that the nodes of the parse-graph are represented by cells that have been extended to include profiling information. To keep track of the relationships between the parse-graph and the source program we assign a unique *colour* for each function to be profiled. Typically, colours will be represented by integers and will be assigned to every cell in the parse-graph. Each cell also contains a 1-bit flag to indicate whether it represents the root of the definition of a profiled function. When the flag is set, the cell is said to contain a *root-marker*.

Colour assignment is performed in two stages. The first stage identifies the roots of the function definitions within the graph:

For each function to be profiled:

(i) locate the root of the subgraph which represents the function definition.
(ii) set the root-marker for this cell.
(iii) assign a unique colour to the cell.

The high-level function names are not present in the executable program; an auxiliary file is therefore generated by the compiler so that the report process can map the colour-coded profile data back to the original functions.

The second stage propagates the colours to the other cells in the graph:

For each cell whose root marker is set:

(i) recursively propagate the colour of the cell to all of its descendants, terminating each branch of the recursion on encountering a cell whose colour is already determined.
Notice that if a profiled function `h` makes a call to an (unshared) unprofiled function `i` then the cells representing the application of `i` will be descendants of `h` and will be given `h`'s colour. This therefore implements inheritance-mode profiling. See below for further discussion of shared unprofiled children.
(ii) if the recursion terminates on a cell whose root-marker is *not* set and whose colour is different from that being propagated then the unprofiled child cell is shared by two (or more) profiled parents. Two methods for dealing with this problem are described below.

4.1 Shared code

Figure 7 illustrates the problem of an unprofiled function which is shared by two or more profiled functions. The graph represents the case when profiles are requested for `f` and `g`, but not for `h`, where `f`, `g`, and `h` are defined as follows:

$\texttt{f} = \texttt{exp}_A\ \texttt{h}$

$\texttt{g} = \texttt{h}\ \texttt{exp}_C$

$\texttt{h} = \texttt{exp}_B$

Two possible alternative compile-time solutions are:

1. Duplicate the cells which represent the shared child, as demonstrated in Fig. 8. If this solution is adopted then the result of the program is unchanged but a loss of sharing occurs and the programmer therefore receives no feedback regarding the beneficial effects of sharing. Furthermore, the overall resource utilization of the program is distorted. We prefer not to implement this solution.

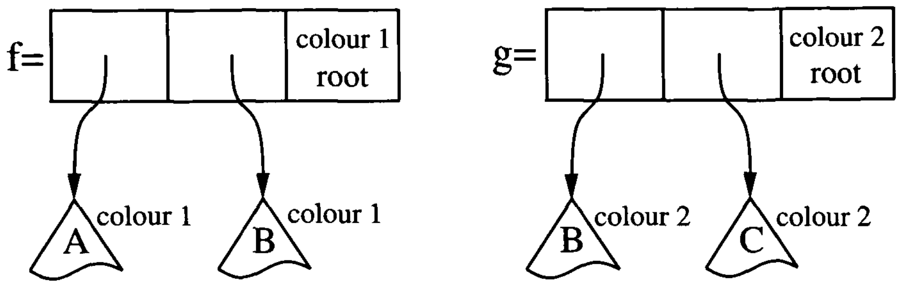

Fig. 8. Duplicating the shared function.

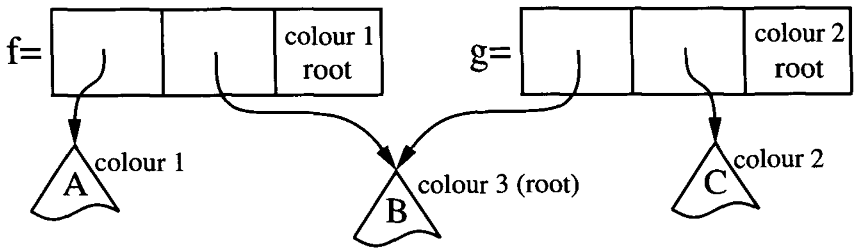

Fig. 9. Profiling the shared function in its own right.

2. Profile the child separately in its own right (see Fig. 9). This may be achieved either by means of a general-purpose *shared-code* colour or, more usefully, by means of a unique colour for each shared, unprofiled function. The advantage of this technique is that the sharing properties of the source program are retained: the programmer can see the effect of using sharing to optimize resource utilization, and the overall resource consumption of the program is not distorted.

Glasgow cost centres provide a run-time solution to the problem of sharing, which they call 'dynamic inheritance'. This has the advantage that shared expressions will only be profiled separately if their profile has been requested (whereas we profile all shared functions separately, regardless of whether this is what the programmer requested). The disadvantages of dynamic inheritance are that it is an additional run-time overhead and that all CAFs must be profiled separately, regardless of what the programmer specified.

4.2 An example of the profiler's compilation phase

Figure 10 shows a graphical representation of the following piece of a program:

```
main  = f 10
f x   = h 1 (g x [1..1000])
g a b = a : reverse b
```

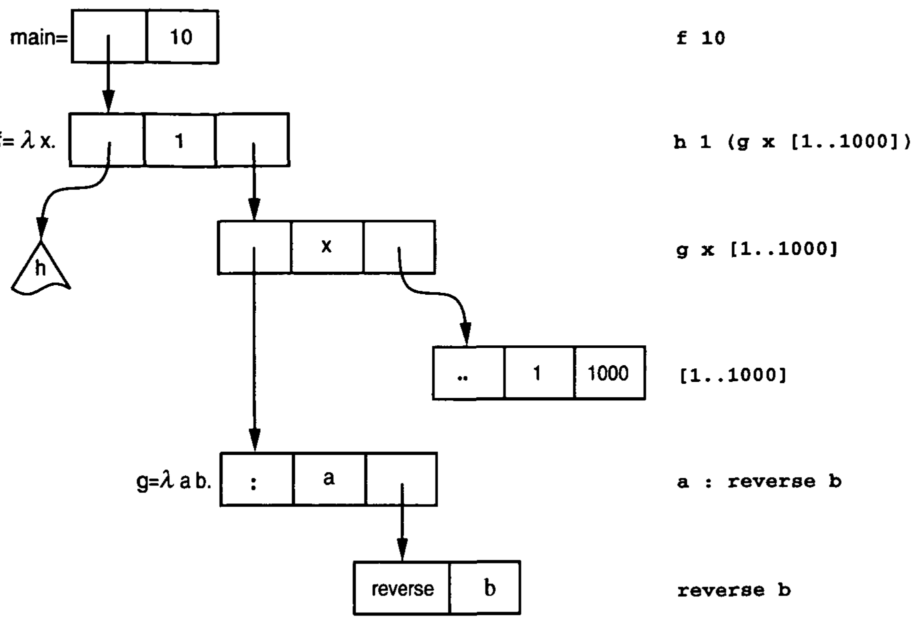

Fig. 10. A graphical representation of the example program.

Functional expressions corresponding to each of the graphical units are given on the right-hand side of the figure.

In the first profiling pass, the compiler extends each of the cells in Fig. 10 by an extra field to accommodate the profiling information and attaches root markers and unique profiling colours to the relevant cells. Figure 11 illustrates the result of this pass for the example program.

In the second profiling pass, the compiler propagates the profile colours to descendant cells in the program graph. The results of this phase are shown in Fig. 12. Notice that there is no problem with unprofiled shared code in this example.

Once every cell has been coloured, any transformations which may subsequently performed by the compiler on the graph must preserve the cell colours so that knowledge of the lexical scoping of the original program is retained. The initial colouring process may occur at an intermediate stage in the compiler (for example, our prototype does colouring at the level of intermediate code) as long as the colours directly relate to the source-level function definitions. Our prototype takes FLIC (Peyton Jones and Joy, 1989) as input and we effect no further program transformations (though we do a substantial amount of static analysis and optimization); thus, the issue of preserving colours in the presence of program transformation is left to further work.

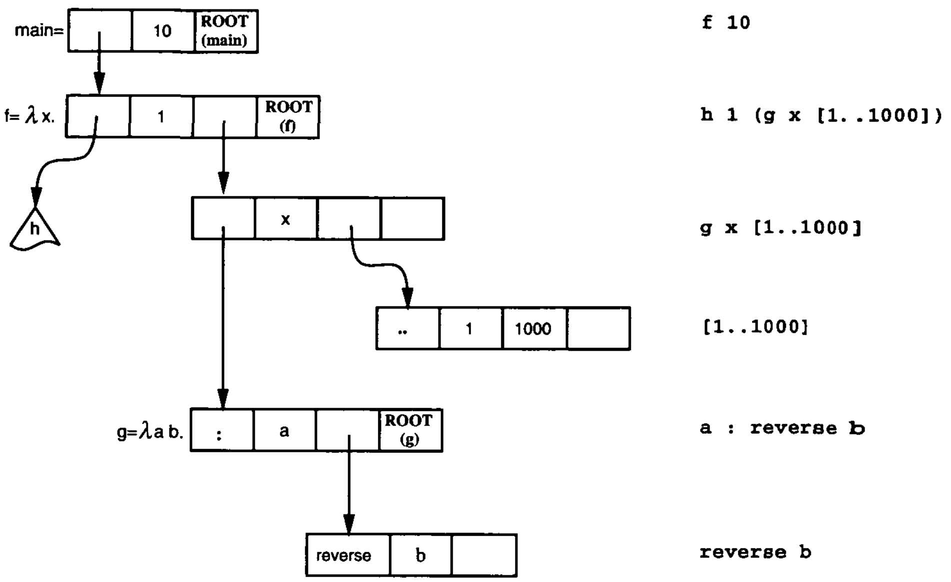

Fig. 11. Introducing root markers and unique root colours.

5 The run-time mechanisms

For straightforward interpretive graph reduction, a program is represented by a graph of binary cells. Each cell consists of left-hand and right-hand fields whose contents are determined by the abstract machine. (Other fields may also be required by the reduction engine to store status information but these have no effect on the profiling mechanism described here.)

5.1 An extended cell representation for profiling

For profiling purposes, each cell will be extended with extra information supplied by the compiler. In this section, we will determine how much extra information is required; we begin with a simple example and show that this does not guarantee to retain the lexical affinities of the source program. We detail two extensions to the simple scheme: the first overcomes the loss of lexical affinities, and the second facilitates enhanced information gathering.

5.1.1 Adding constructor information to cells

A naïve attempt at lexical profiling is shown in Fig. 13. The graph represents the following program segment:

```
let g x = x exp_A exp_B
    h   = g f
in h
```

Each graph cell is augmented with the colour of the function which constructed it in

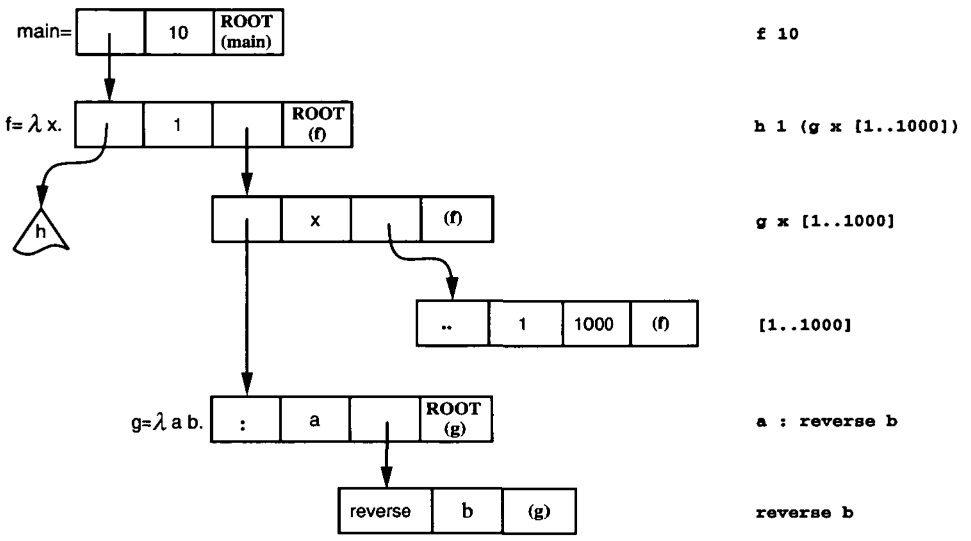

Fig. 12. Propagating profile colours downwards from the roots.

the same manner as the compiler's parse-graph shown in Figs. 10, 11, and 12. We shall henceforth use the terms *constructor-function* and its associated *constructor-colour* (though the reader should take care not confuse these terms with Runciman and Wakeling's 'constructor-profiles'). The graph segment shown in part (ii) of Fig. 13 is the result of instantiating g with the argument field, ⓐ, using template instantiation (Peyton Jones and Lester, 1992, Ch.2). The redex has been overwritten by the result and the movement of key fields from graph segment (i) to graph segment (ii) is illustrated by the labels ⓐ, ⓑ, and ⓒ.

There are several points of interest in this example. Firstly, notice that the constructor-colour, h, of the overwritten redex has not been updated. This is explained in more detail in section 5.4. Secondly, notice that in part (i) of the figure, prior to instantiating g, the reference to function f was contained within a cell whose constructor-colour was given as *h* but in part (ii) of the figure, after the instantiation, the reference is contained within a *new* cell whose constructor-colour is given by g. This clearly contravenes the rules of lexical profiling because it now appears as though the reference to f occurred lexically within g.[‖]

5.1.2 Retaining lexical affinities

The above problem is due to the unboxed argument, ⓐ, and might naïvely be solved by ensuring that all argument values are boxed. Figure 14 demonstrates this principle. In part (i) of the figure, prior to the instantiation of g, the argument containing the reference to f is boxed and thus occupies an extra cell. After the

[‖] See section 5.1.5 for a comparison between this problem and the current behaviour of Glasgow cost centres.

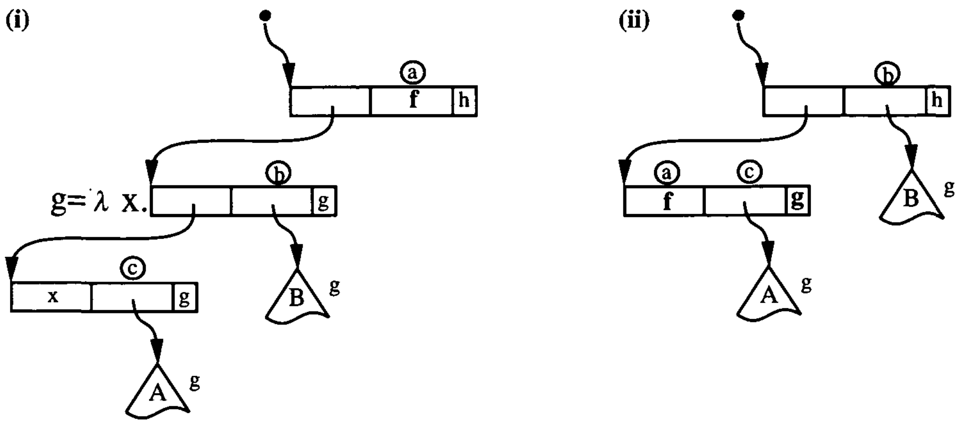

Fig. 13. A naïve attempt at profiling lazy, higher-order functions.

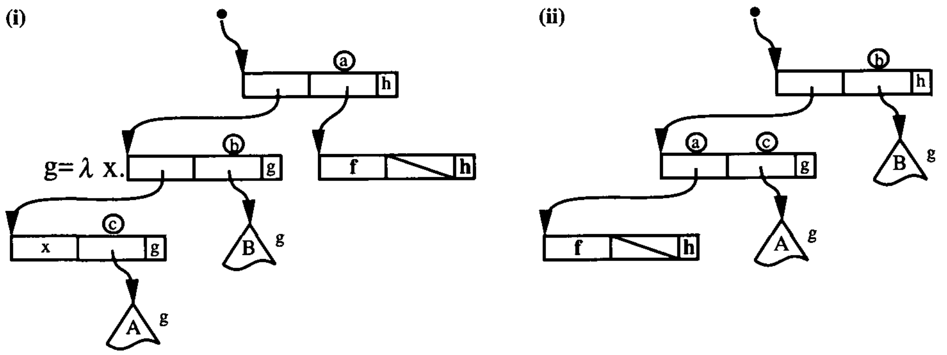

Fig. 14. Lexical profiling using boxed arguments.

instantiation (part (ii) of the figure), the cell is unchanged, hence maintaining a constant constructor-colour for the reference to `f`.

The method of boxed arguments provides correct lexical profiling information but requires the reduction engine to translate *all* unboxed arguments into boxed values. This will need changes to be made to the way that most abstract machines perform graph reduction and is extremely inefficient. Furthermore, this will distort the space utilization of the program. Ideally, the profiling mechanism should only require changes that relate directly to profiling and not to the reduction strategy.

Fortunately the problem can be overcome by assigning additional colouring information to the left-hand field, L, and to the right-hand field, R, of each cell. Since L and R each have their own profiling information, values can be safely unboxed without losing vital information.

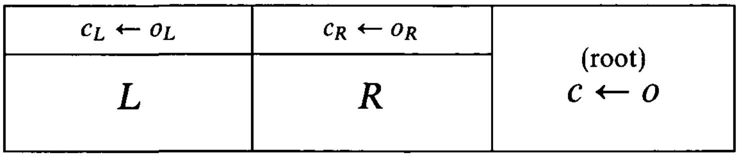

Fig. 15. A fully augmented graph cell.

5.1.3 Identifying reference sites

With the above scheme, we may determine the degree to which functions are responsible for consuming time and space; however, lexical profiling requires that the costs for a function are categorized according to the different reference sites of that function in the source program.

There are many ways in which this categorization may be achieved. For example, the user might explicitly identify which reference sites are of interest and give them unique names (Sansom and Peyton Jones, 1992) which would be used to define the cell and field colour information. However, we chose to implement a different method which automatically identifies the different reference sites. This has two advantages:

1. The user is spared the effort of identifying the reference sites.
2. Where a function is degenerate only at a subset of its reference sites, the user does not need to know in advance which reference sites to monitor.

The two disadvantages of our approach are:

1. The user may be provided with more information than necessary; however, we provide the user with the facility to monitor specific functions, thereby reducing the total amount of information gathered.
2. If more than one reference of the same function is made in a single expression, our method will not distinguish between them.

To categorize function reference sites automatically, we redefine a colour to be a two-colour pair denoted by $c \leftarrow o$. The $\leftarrow$ operator is used to combine the two colours of each pair into a single, compound symbol; we use a $\leftarrow$ symbol as a metaphor for lexical containment or derivation (thus $c \leftarrow o$ means that 'this application of c is lexically contained in the definition for o'). The colour c is the constructor colour, whereas the colour o represents the reference site. Each graph cell and each of its fields is extended with a colour pair, as depicted in Fig. 15.

The c, c_L, and c_R colours indicate the constructor-function for the whole cell, the item in the left-hand field, and the item in the right-hand field, respectively. For the cells of supercombinator templates, $c = c_L = c_R$. These are assigned statically when the program is loaded, according to the colouring information supplied by the compiler. When supercombinator templates are instantiated (see sections 5.2 and 5.3), a mutable copy of the template graph is constructed. The *fields* which contained formal parameters in the template are instantiated with both the value and the profiling colours of the actual parameters. Therefore, in the mutable part of the program graph, the c, c_L, and c_R colours within a single cell may be different.

The colours o, o_L, and o_R are *origin-colours* and are used to record which functions lexically contain references to the corresponding constructor-functions in the source program. Profiling information can then be reported back to the programmer in terms of the lexical function origins present in the source code. Consider the example program given at the start of section 3. Here, the origin-colours of cells constructed by `map` will be set to the colour of function `g` for the first call to `map` and to the colour of function `h` for the second call. Again, the o, o_L, and o_R colours may differ from each other when either or both of the left- or right-hand fields are instantiated by actual parameters. Since origin-colours are set during instantiation, they are not defined for the cells of supercombinator templates.

5.1.4 Summary of colouring information

In total, we use six colours plus a 1-bit root-node marker which is set in the top cell of profiled functions and reset in all other cells (see section 4). The six colours are arranged into three pairs: $c \leftarrow o$ (defined for the whole cell), $c_L \leftarrow o_L$ (defined for the left-hand field), and $c_R \leftarrow o_R$ (defined for the right-hand field). Table 3 provides a summary of the colouring information (where $\neq$ means 'is not necessarily equal to').

5.1.5 Comparing the six-colour scheme with Glasgow cost centres

We have now shown how *full* lexical profiling can be achieved using a six-colour scheme. At this point it is worthwhile comparing our profiling scheme more closely with its nearest alternative, Glasgow's cost centres (Sansom and Peyton Jones, 1992).

Cost centres employ a colouring scheme similar to that described in section 5.1.1. Thus, only a single colour is required per closure. We have already described in some detail the problems that are encountered with this scheme in the presence of higher-order functions. In the defence of the single-colour mechanism, however, it can be argued that functions really have zero cost and that it is their *application* which consumes resources. If this argument is accepted then the lexical position of a function applied to zero arguments is irrelevant, and the single-colour scheme is acceptable. However, the eventual application of the function to its arguments will, as a consequence, be determined dynamically. We believe that this contradicts the spirit of lexical profiling because it is difficult for the programmer to make sense of the profiling results in the context of the source code for the program. Furthermore, when the function whose cost is assumed to be zero is actually a CAF, the single-colour scheme may give incorrect results. The CAF may require some evaluation, hence its *non-zero* cost should be attributed lexically, not dynamically. To avoid the problems of attributing the cost of inherited CAFs, cost centres always profile CAFs separately, regardless of whether or not they are shared. Furthermore, in recognition of the problem of attributing the costs of higher-order functions, cost centres have now been adapted so that a program transformation is applied to every top-level

Table 3. *A summary of colouring information.*

Colour	Description
c	Identifies the function responsible for creating the current graph cell.
o	Identifies the origin of the function responsible for creating the current graph cell. From this colour we can determine where the function reference was made, lexically within the source code.
c_L	Indicates which function originally created the left-hand field of the current cell. This is needed to keep track of the constructor colour of functions that have been passed as arguments. If the field contains a higher-order argument then $c_L \neq c$. This colour is used in section 5.2 to determine the origin colour of a profiled supercombinator when the supercombinator is instantiated.
o_L	The origin colour of the left-hand field of the current cell. As for c_L, if the field contains a (higher-order) argument then $o_L \neq o$. This colour is used in section 5.3 to determine the origin colour of a *non*-profiled supercombinator when the supercombinator is instantiated.
c_R	The constructor colour of the right-hand field of the current cell. Again, if the field contains a higher-order argument then $c_R \neq c$. This colour is needed when the value of the field is passed as an actual parameter to another supercombinator where it is subsequently applied as a function to some arguments.
o_R	The origin colour of the right-hand field of the current cell. As for c_R, if the field contains a higher-order argument then $o_R \neq o$ and, again, the colour is needed when the value of the field is passed as an actual parameter to another supercombinator.

inherited function which is passed as a higher-order function: this effectively 'boxes' the higher-order function to capture the cost centre of its reference site.

The six-colour scheme *always* attributes the cost of applying a function (*including* CAFs) to the position in the source code where the function was referenced. It is a simple scheme which does not require 'boxing' transformations and does not require special treatment for CAFs.

5.2 Instantiating profiled supercombinators

If the root marker in the top cell of the supercombinator's template is set, then the template represents a profiled function. Figure 16 illustrates the procedure for instantiating profiled functions, using the supercombinators f_1 and f_2 which are defined as follows:

$$f_1 \quad = f_2 \; exp_A$$
$$f_2 \; x = exp_B \; x$$

Consider part (i) of the figure which represents the state of the reduction immediately prior to the instantiation. The o, o_L, and o_R colours of every cell in f_2's

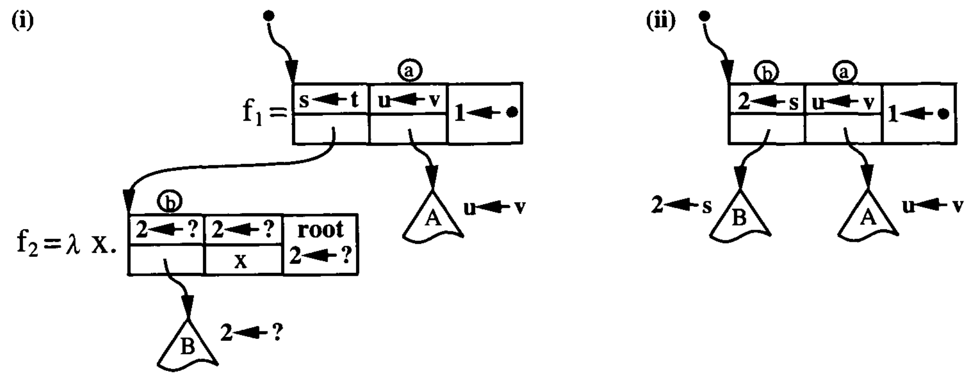

Fig. 16. Instantiating a profiled supercombinator.

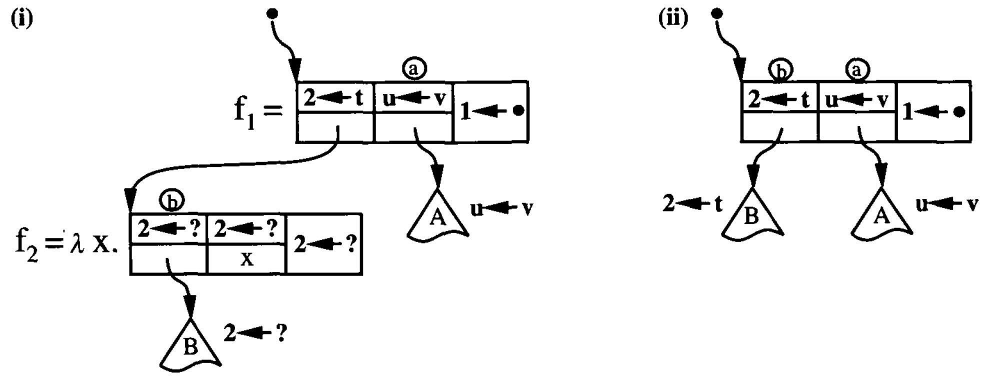

Fig. 17. Instantiating an unprofiled supercombinator.

template are as yet undefined. In this case, the origin of the reference to f_2 is given by the constructor-colour, $c_L = s$, in the left-hand field of the cell which points to the function f_2 (in this example, it happens to be the left-hand field of the root cell of the redex). That is therefore the value assigned to the origin-colours of the instantiated graph shown in part (ii) of the figure. Notice that, as before, the colours $c \leftarrow o$ of the overwritten redex remain unchanged and that the actual parameter, ⓐ, retains its colours, $u \leftarrow v$.

5.3 Instantiating unprofiled supercombinators

Figure 17 illustrates the procedure for instantiating a supercombinator whose root marker is *not* set. This uses the same supercombinator definitions as Fig. 16 but represents a call to a function, f_2, that is not being profiled in its own right. For the purposes of profiling, f_2 has been subsumed into its origin-function, f_1. The instantiation therefore differs from that of profiled supercombinators in two ways. Firstly, the constructor-colour in the left-hand field of the redex will *always* be identical to that of the subsumed supercombinator. Secondly, the origin-function for the subsumed supercombinator is the same as the origin-function of the profiled function into which it was subsumed. Therefore, the origin-colour of each new cell generated by the instantiation of $(f_2 x)$ is determined by the *origin-colour*, $o_L = t$, in the left-hand field of the root node of the redex. All other aspects of the instantiation are identical to the profiled case.

5.4 Profiling functions

In this section we present the techniques used for profiling functions. Our techniques make almost no distinction between the profiling of functions and the profiling of CAFs. However, other profilers (such as the Glasgow cost centres) find that CAFs present particular difficulties and so the next section will specifically address the profiling of CAFs.

Call-count profiling

The number of calls to a profiled function is determined by the number of times its associated supercombinator is instantiated. Sections 5.2 and 5.3 described two types of instantiation, the first of which deals with *profiled* supercombinators. Call-counts are incremented only for *profiled* supercombinators.

A separate call-count register is maintained for every $c \leftarrow o$ combination, where c is statically bound to the cells of the supercombinator template and o is determined by the method described in section 5.2.

Space profiling

Space profiling uses the profiling colours $c \leftarrow o$ attached to the cells in the program graph to determine the total number of cells allocated for a function and the maximum number of cells allocated at any one time. Space profiling information is recorded separately for each $c \leftarrow o$ combination and requires only small changes to be made to the cell allocation and garbage collection code so that the relevant registers are updated each time a cell is allocated or deallocated. To monitor continuously the number of heap cells that are currently active we assume reference-count garbage collection (Glaser *et al.*, 1988; Hughes, 1987; Axford, 1990; Hudak, 1986; Lermen and Maurer, 1986; Shute, 1988). If a different style of garbage collection were used, such as two-space copying (Baker, 1978; Rudalics, 1986) or mark/sweep (Cohen, 1981; Hughes, 1985) then the number of active heap cells could only be approximated by an upper bound. This would only be accurate immediately after a garbage collection has taken place and would steadily lose accuracy as cells become inactive, up to the next collection (see also section 2.6).

In classical graph reduction the root node of the redex is overwritten with the result after each reduction has taken place to ensure that shared values are not recomputed (there are many and varied discussions about this in, for example, Johnsson, 1984; Augustsson, 1984; Fairbairn and Wray, 1987; Peyton Jones, 1987a, b; Burn *et al.*, 1988; Peyton Jones and Salkild, 1989; Augustsson and Johnsson, 1989a, b). In particular, we note that the result of a reduction may not require any cell allocation (for example, consider the identity function) and so the above sharing requirement means that many graph reducers transform the root node of the redex into an *indirection* node (Peyton Jones, 1987b). This indirection node may be garbage collected once all the pointers to the node have been dereferenced – until that time, however, the indirection cell will remain part of the live heap and it seems reasonable to attribute this space cost to the function which first allocated the cell (since the space cost is a direct result of the sharing of that function).

Thus, for space profiling purposes we must take care when overwriting a redex to leave the profiling colours of the cell intact (the profiling colours for the fields may change of course), otherwise the de-allocation of the cell by the garbage collector will decrement a different space-usage register to that which was incremented when the cell was allocated. The overwriting of the root node of the redex with an indirection has the added benefit that the space occupied by a data structure will be attributed to the function which constructed it rather than to the function which allocated the root cell of the redex. The colours attached to the fields of the updated redex ensure that the higher-order lexical profiling continues to function correctly.

Time profiling

The implementation of time profiling is closely tied to graph traversal. In our simple interpretive reduction engine, graph traversal is restricted to the operation of *unwinding* the current spine of the graph in search of the next function to apply. The profiler maintains a separate timing for each $c_L \leftarrow o_L$ colour combination and makes use of a *current* $c_L \leftarrow o_L$ register. We use the colours of the left-hand field because the spine of the graph is encoded there and it is the pointers in the spine which determine function calls. Notice, however, that when a *rib* (Clack and Peyton Jones, 1986) is evaluated, the *current* $c_L \leftarrow o_L$ register will take its value from the redex of the rib, which occupies the right hand field of a graph cell.

At the start of the reduction all timers are zeroed, the current $c_L \leftarrow o_L$ register is set to the $c_L \leftarrow o_L$ of the initial redex, and the system time is read and stored in t. Reduction then proceeds as normal by unwinding the spine. When unwind encounters a cell whose $c'_L \leftarrow o'_L$ colour differ from the current $c_L \leftarrow o_L$:

(i) The current system time t' is read.
(ii) The accrued time for $c_L \leftarrow o_L$ is incremented by the elapsed time $t' - t$.
(iii) The timer t is updated with t'.
(iv) The current $c_L \leftarrow o_L$ register is set to $c'_L \leftarrow o'_L$.

The unwind operation is performed each time a redex is overwritten. As long as the unwind always starts at the overwritten redex, no further action is required to profile time costs. This mechanism provides the required time, space, and call-count profiling information even when lazy evaluation causes the graph to become fragmented and results in many context switches.

Garbage Collection costs are *not* included in function time costs; it is a simple matter to temporarily suspend timings while garbage collection occurs. However, it is in general a subtle matter to decide how much storage allocation (and deallocation) cost should be attributed to the function and how much to the storage manager. A function should be charged for the time it takes to claim or free a cell, but should not be charged with the costs of general storage management which benefits all functions**.

5.5 *Profiling CAFs*

Constant Applicative Forms (CAF) are not necessarily instantiated in the same way as normal functions because they have no arguments. If a top-level identifier is a CAF then our profiler allows it to be profiled and the CAF is coloured in the same way as a normal function would be coloured. The only way in which a profiled CAF is treated differently is that supaercombinator instantiation cannot be used as the trigger for incrementing the call-count. In our implementation a profiled CAF is detected when the *unwind* operation (Peyton Jones, 1987b) passes through a cell whose root marker is set. The appropriate $c \leftarrow o$ call-count register for the associated CAF is then incremented: just as with profiled supercombinators, c is statically bound to the cells of the CAF and o is determined by the method described in section 5.2.

Space profiling of CAFs is carried out in exactly the same way as space profiling of any other function. A CAF is charged for the space used in its graph representation, including any embedded data structures, and any space utilized by subsumed, unprofiled functions: space utilized by any profiled functions referenced by the CAF will not be charged to the CAF. Embedded data structures may not reach their final form unless the CAF is applied to other arguments, but any space utilized is still charged to the CAF. Similarly, subsumed unprofiled functions may not require instantiation unless the CAF is applied to other arguments, but any space utilized is charged to the CAF.

Time profiling of CAFs is achieved in the same way as the time profiling of any other piece of graph. Time is attributed to the CAF when the graph cells for the CAF are traversed, when embedded data structures are evaluated, and when subsumed unprofiled functions are instantiated.

5.6 *Accuracy and overheads of the technique*

The above technique for measuring the time utilized by functions is only as accurate as allowed by the system clock, and cumulative errors may be large for an interpretive reducer.

Typically the system clock will provide time to the nearest 0.02 seconds, though

** In general, the implementor of a profiling system will find that reference counting costs are easy to attribute but hard to factor out and report separately, whereas stop and copy costs are easy to factor out, but harder to attribute.

some modern UNIX workstations provide higher-resolution timers (for example, 2 or 4 times higher resolution). Furthermore, the overhead of inspecting the system clock may differ from system to system. The large number of start/stop measurements will combine the timing inaccuracies and will lead to correspondingly large worst-case errors – however, the *expected* errors in the measured times are much more reasonable, as can be verified by implementing a statistical error package which, for example, convolves the probability density functions for each measurement. We have incorporated error analysis into our prototype profiler, but this is not yet fully debugged and so we leave the reporting of expected errors to a subsequent paper.

The lexical profiling technique also imposes an overhead on execution time. This leads to a problem of self-reference, in that a program's computational costs may become distorted because of the presence of the profiling mechanism. We would like to be able to demonstrate that the overheads of lexical profiling are in linear proportion to the costs of unprofiled execution. Informally, this may be argued from the observation that no new abstract machine instructions are added, and that where an abstract machine instruction is extended then the overhead in space and time is always a (small) constant. However, if the modified instructions are executed more often than the unmodified instructions then this may still lead to a distortion in the computational costs. There is little point in attempting an analytic approach to this problem, since there is no theoretical basis on which to calculate the proportionate use of individual abstract machine instructions. Rather, experimental observation must be employed.

Our prototype profiler uses an interpretive graph reducer as a simple demonstrator. Experimental results of profiling overheads for interpretive graph reduction are of little use to the research community (since interpretive graph reduction was long ago superseded by compiled graph reduction) and so we defer our experimental analysis of overheads until we have implemented profiling for a compiled reducer. We have already started this work and we will report our results in a subsequent paper.

It is possible to use sampling (Sansom and Peyton Jones, 1992; Graham *et al.*, 1982) of the current $c_L \leftarrow o_L$ register in order to minimize the cumulative errors and to reduce the start/stop timing overheads, but at the cost of possible loss of information if the sampling frequency is too large. This does not detract from the essential nature of lexical profiling, but merely incurs measurement errors in a different (and possibly lesser) way to those incurred by the start/stop technique.

5.7 Single-stepped example of lexical profiling

In this section the steps of the reduction shown in Fig. 18 are explained with specific reference to lexical profiling activities. The figure is based on the following function definitions:

```
h   = g f
g x = x exp_A
f y = + exp_B y
```

and the explanation relies on the profiling terms described in the previous sec-

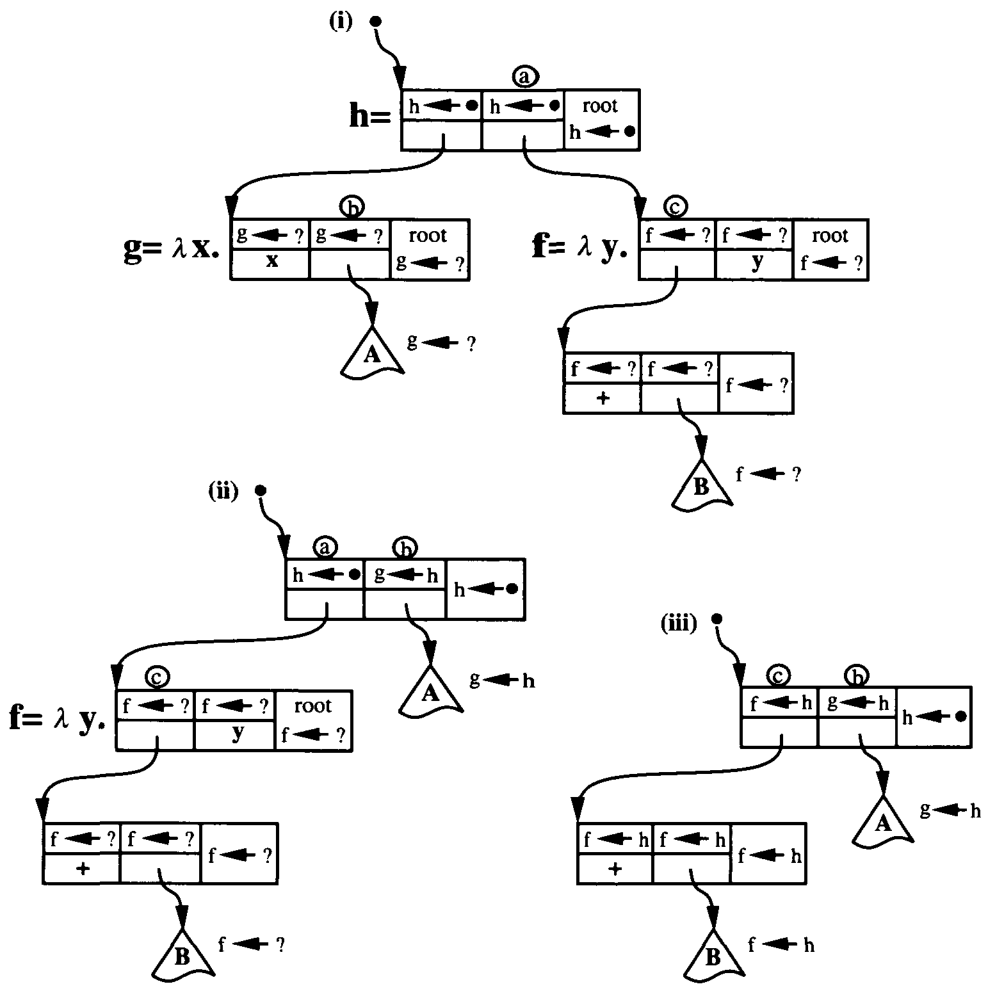

Fig. 18. Lazy, higher-order profiling.

tions; the reader should be familiar with these before continuing. In this example, the root of the program is denoted by the • symbol.

Part (i)

1. The current $c_L \leftarrow o_L$ register is set to $h \leftarrow \bullet$ and the spine is unwound (the time for $h \leftarrow \bullet$ is *not* yet updated).
2. The profiled function, g, is encountered and the origin-colour for the function is determined from the left-hand field of h where $c_L = h$.

Part (ii)

1. The redex is overwritten with the instantiated copy of g (the cell allocator increments the space usage for $g \leftarrow h$ as the new cells are allocated). The formal parameter, x, has been overwritten by the argument field, ⓐ, complete with its profile colours, $h \leftarrow \bullet$.
   The origin-colour of the reference to g is set to h as determined in part (i). It is written into o_R in field ⓑ and into o, o_R, and o_L of the cells in the subgraph, A. The latter requires a graph traversal of A, but we note that this is required anyway since A is part of the function template and must be copied.
2. The call-count for $g \leftarrow h$ is incremented.
3. The spine is unwound a second time (again, the timers are *not* yet updated).
4. The profiled function, `f`, is encountered and the origin-colour for the function is determined from the field ⓐ where $c_L = h$. This is correct because the reference to `f` originated from `h` in part (i).

Part (iii)

1. The instantiation proceeds as before. The redex is overwritten with the instantiated copy of `f` (the cell allocator adjusts the space-usage for $f \leftarrow h$ as the cells are allocated). The argument field, ⓑ, overwrites the formal parameter and the origin-colours in and below field ⓒ are set to h.
2. The call-count for $f \leftarrow h$ is incremented.
3. The spine is unwound for a third time and, since the $f \leftarrow h$ colours of the left-hand field differ from the current $c_L \leftarrow o_L$ register value of $h \leftarrow \bullet$, the elapse timing is adjusted for $h \leftarrow \bullet$. The current $c_L \leftarrow o_L$ register is set to $f \leftarrow h$ and the unwind continues until the primitive function + is reached.
4. At this point the values of A and B are required. Each is reduced in turn before the + function can be executed. Notice that the profile colours of A and B correctly identify their constructor- and origin-functions, hence the technique has behaved correctly in the presence of lazy evaluation.

6 Operational semantics

The following operational semantics for lexical profiling assumes the existence of an operational semantics $\mathcal{E}$ for an interpretive graph reducer. The full definition of $\mathcal{E}$ is beyond the scope of this paper but, for lexical profiling purposes, we assume that $\mathcal{E}$ possesses the following clauses:

$\mathcal{E}_U$ G describing the action taken by the reducer immediately *after* a pointer in the spine of the graph has been followed during an `unwind` operation.

$\mathcal{E}_A$ G describing the action taken by the reducer immediately *after* moving to the root of an argument redex to evaluate that argument, but *prior* to the evaluation.

$\mathcal{E}_I$ G describing the action taken by the reducer to instantiate a supercombinator.

We have assumed that all three of these take a single state variable, G, which describes the entire state of the reduction, including: the graph being reduced, the supercombinator templates, and any stacks that may be required.

To describe the action of the lexical profiler, $\mathscr{E}$ is superseded by $\mathscr{E}'$. Except for the three clauses listed above, each clause of $\mathscr{E}$ has a direct equivalent in $\mathscr{E}'$ that is expanded with the following items of state:

$\langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, \mathbf{root} \rangle$ which represents the state of the *current* cell in the graph. The components of the tuple are described as follows:

- L is the left-hand field of the cell
- R is the right-hand field of the cell
- $c_L \leftarrow o_L$ is the constructor and origin colour for the left-hand field
- $c_R \leftarrow o_R$ is the constructor and origin colour for the right-hand field
- $c \leftarrow o$ is the constructor and origin colour for the whole cell
- **root** is set to 1 if the cell represents the root of a profiled function, and is set to 0 otherwise.

$\{c_C \leftarrow o_C\}$ represents the value of the *current* $c_L \leftarrow o_L$ register.

τ is the set of timings for all $c \leftarrow o$ pairs. We define the semantic function $\tau(c \leftarrow o)$ to return the timing for a specific $c \leftarrow o$ pair and the syntax $\tau[c \leftarrow o \mathrel{+}= s]$ to indicate an increment of s seconds to that timing within the set of timings.

μ is the set of counters which keep track of memory-usage for all $c \leftarrow o$ pairs. We define the semantic function $\mu(c \leftarrow o)$ to return the memory-usage for a specific $c \leftarrow o$ pair and the syntax $\mu[c \leftarrow o \mathrel{+}= m]$ to indicate an increment of m graph cells to that memory-usage within the set of counters.

κ is the set of counters which monitor the number of function invocations. A separate counter is stored for each $c \leftarrow o$ pair, indicating the number of function calls as measured for the corresponding arc in the lexical call graph of the program. We define the semantic function $\kappa(c \leftarrow o)$ to return the call-count for a specific $c \leftarrow o$ arc, and the syntax $\kappa[c \leftarrow o \mathrel{+}= 1]$ to indicate a unit increment to that call-count within the set of counters.

t which records the time at which the *current* $c_L \leftarrow o_L$ register was last set.

The three clauses $\mathscr{E}_U$, $\mathscr{E}_A$, and $\mathscr{E}_I$ are replaced by $\mathscr{P}_U$, $\mathscr{P}_A$, and $\mathscr{P}_I$, respectively. These are defined in Figure 19. Notice that $\mathscr{P}_U$, $\mathscr{P}_A$, and $\mathscr{P}_I$ refer to clauses $\mathscr{E}'_U$, $\mathscr{E}'_A$, and $\mathscr{E}'_I$. As expected, these behave exactly as $\mathscr{E}_U$, $\mathscr{E}_A$, and $\mathscr{E}_I$ except that they are extended with the state items described above. $\mathscr{P}_0$ is used for initialisation.

7 Experience gained using the lexical profiler

This section shows the results obtained from the UCL lexical profiler. Our inspiration for the lexical profiler originated in PhD research involving large functional applications (Clayman, 1993) and we are strongly motivated to continue the development of our prototype profiler so that it is effective with such large applications. In our preliminary studies we have run the profiler on a number of programs ranging from small, familiar benchmarks such as the `nqueens` problem to a relational

$$\begin{array}{lll}
\mathscr{P}_0 \ \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, r\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t & & \\
\quad = \mathscr{E}'_0 & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, r\rangle & \\
 & \{c_L \leftarrow o_L\} \ \tau_0 \ \mu_0 \ \kappa_0[c_L \leftarrow o_L = r] \ G \ t_c &
\end{array}$$

$$\begin{array}{lll}
\mathscr{P}_U \ \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 1\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t & & \\
\quad = \mathscr{E}'_U & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 1\rangle \ \{c_L \leftarrow o_L\} & \\
 & \tau[c_C \leftarrow o_C \mathrel{+}= (t_c - t)] \ \mu \ \kappa[c \leftarrow o \mathrel{+}= 1] \ G \ t_c, & \text{if } (c_L \leftarrow o_L) \neq (c_C \leftarrow o_C) \\
\quad = \mathscr{E}'_U & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 1\rangle \ \{c_C \leftarrow o_C\} & \\
 & \tau \ \mu \ \kappa[c \leftarrow o \mathrel{+}= 1] \ G \ t_c, & \text{if } (c_L \leftarrow o_L) = (c_C \leftarrow o_C)
\end{array}$$

$$\begin{array}{lll}
\mathscr{P}_U \ \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 0\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t & & \\
\quad = \mathscr{E}'_U & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 0\rangle \ \{c_L \leftarrow o_L\} & \\
 & \tau[c_C \leftarrow o_C \mathrel{+}= (t_c - t)] \ \mu \ \kappa \ G \ t_c, & \text{if } (c_L \leftarrow o_L) \neq (c_C \leftarrow o_C) \\
\quad = \mathscr{E}'_U & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 0\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t, & \text{if } (c_L \leftarrow o_L) = (c_C \leftarrow o_C)
\end{array}$$

$$\begin{array}{lll}
\mathscr{P}_A \ \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 1\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t & & \\
\quad = \mathscr{E}'_A & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 1\rangle \ \{c_L \leftarrow o_L\} & \\
 & \tau[c_C \leftarrow o_C \mathrel{+}= (t_c - t)] \ \mu \ \kappa[c \leftarrow o \mathrel{+}= 1] \ G \ t_c, & \text{if } (c_L \leftarrow o_L) \neq (c_C \leftarrow o_C) \\
\quad = \mathscr{E}'_A & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 1\rangle \ \{c_C \leftarrow o_C\} & \\
 & \tau \ \mu \ \kappa[c \leftarrow o \mathrel{+}= 1] \ G \ t_c, & \text{if } (c_L \leftarrow o_L) = (c_C \leftarrow o_C)
\end{array}$$

$$\begin{array}{lll}
\mathscr{P}_A \ \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 0\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t & & \\
\quad = \mathscr{E}'_A & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 0\rangle \ \{c_L \leftarrow o_L\} & \\
 & \tau[c_C \leftarrow o_C \mathrel{+}= (t_c - t)] \ \mu \ \kappa \ G \ t_c, & \text{if } (c_L \leftarrow o_L) \neq (c_C \leftarrow o_C) \\
\quad = \mathscr{E}'_A & \langle L, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 0\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t, & \text{if } (c_L \leftarrow o_L) = (c_C \leftarrow o_C)
\end{array}$$

$$\begin{array}{ll}
\mathscr{P}_I \ \langle \mathrm{SC}\ y, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 1\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t & \\
\quad = \mathscr{E}'_I & \langle \mathrm{SC}\ y', R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 1\rangle \\
 & \{c_C \leftarrow o_C\} \ \tau \ \mu[x \leftarrow s \mathrel{+}= |SC\ y|] \ \kappa[c \leftarrow o \mathrel{+}= 1] \ G \ t \\
 & \text{where } (s \leftarrow t) = (c_L \leftarrow o_L) \\
 & y' = y[x \leftarrow s / c_L \leftarrow o_L; \ x \leftarrow s / c_R \leftarrow o_R; \ x \leftarrow s / c \leftarrow o] \\
\mathscr{P}_I \ \langle \mathrm{SC}\ y, R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 0\rangle \ \{c_C \leftarrow o_C\} \ \tau \ \mu \ \kappa \ G \ t & \\
\quad = \mathscr{E}'_I & \langle \mathrm{SC}\ y', R, c_L \leftarrow o_L, c_R \leftarrow o_R, c \leftarrow o, 0\rangle \\
 & \{c_C \leftarrow o_C\} \ \tau \ \mu[x \leftarrow t \mathrel{+}= |SC\ y|] \ \kappa \ G \ t \\
 & \text{where } (s \leftarrow t) = (c_L \leftarrow o_L) \\
 & y' = y[x \leftarrow t / c_L \leftarrow o_L; \ x \leftarrow t / c_R \leftarrow o_R; \ x \leftarrow t / c \leftarrow o]
\end{array}$$

Fig. 19. Operational semantics for lexical profiling.

database of several hundred lines of code. We have not yet had the opportunity to use the profiler on a large application, but the examples in this section will provide a guide to the kind of results which may be obtained.

We present profiling results for three programs:

1. The first program is a small, specially-constructed example which will demonstrate how the lexical profiler may be used to detect a common programming problem: a *pipeline blockage.*
2. The second program is the familiar `nqueens` problem, which demonstrates that it is important for a profiler to measure *time* consumption of a program as well as heap space consumption.

3. The final program investigates the different resource consumptions of two styles of programming: the composition of standard higher-order functions (which encapsulate recursion) versus the use of explicit recursion. This example uses the lexical profiler to demonstrate the tradeoff between time and space.

For our experiments we have used the UCL experimental interpreting reducer (rather than an optimized compiled reducer) and space usage is reported in terms of heap cells (rather than bytes) since we believe that a cell count is more informative than a byte count for an implementation which uses cells with a fixed number of fields (the sizes of cells may vary from run to run according to different compiler and profiler options and a byte count would therefore require more work from the user when making comparisons between runs, though we appreciate that a byte count would probably be required for an implementation with a variable number of fields per cell). The profiling data for time and call-count is presented separately from the space usage data and execution times are accumulated and reported for every profiled function. The statistics for each function are subdivided according to the functions that called it, and times denote the actual execution time rather than the elapsed wall-clock time. The time for (reference count) garbage collection is determined separately and is *not* included in the time for any function. The profiler produces tabular data for call-count and timing: this data has been post-processed and is presented as vertical bar-charts.

7.1 Detecting a pipeline problem

In Runciman and Wakeling (1993), significant improvements are made to a program called `clausify` which calculates the clausal forms of a series of propositional formulae. The improvements are the result of a repeated process of profiling and program modification. During the exercise a *pipelining* problem was discovered and, by reasoning from the heap-profile, the programmers were led to notice the abberant behaviour of a partial application of the `foldr` function.

It was intended that a list of values would be generated lazily using `foldr` and that the result would be consumed lazily by a pipeline of processing functions. However, using a heap profiler it was possible to observe that the whole list was being constructed strictly as soon as its first value was needed. As the list was large, it occupied a large area of memory; when the problem was solved by making the program less strict, the list was constructed and garbaged incrementally.

The reported problem was due to *tail-strictness*. This is important because functions that are unexpectedly tail-strict will construct whole lists even if only a portion of the result is ultimately to be consumed. Therefore, the issue of increased heap-occupancy is compounded by wasted processing effort.

The problem is not, however, inherent in the `foldr` function because `foldr` is *not* tail-strict. The tail-strictness property is introduced by the higher-order parameter. Consider the following Haskell program which illustrates an extreme example of this problem:

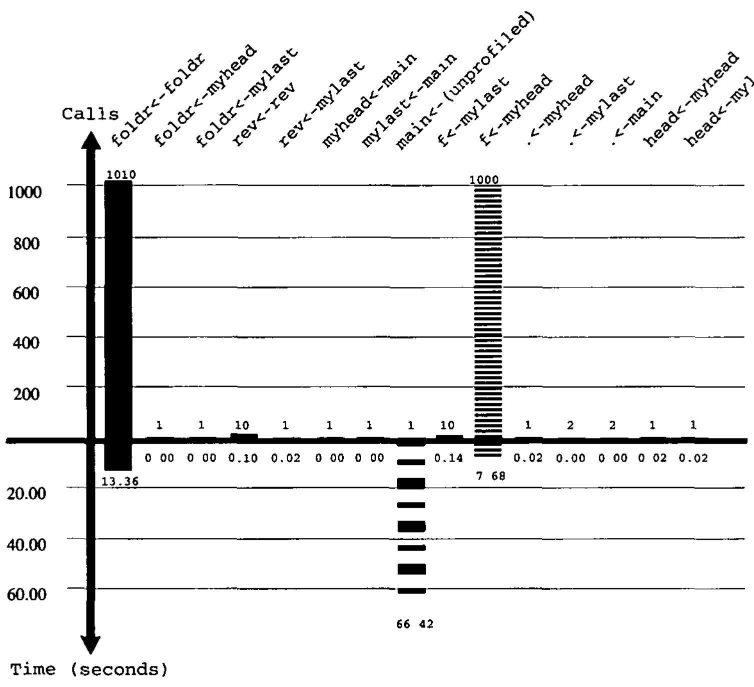

Fig. 20. Call counts and time profile for a blocked pipeline.

```
f x []      = x:[]
f x (y:ys)  = x:y:ys

rev :: [a] -> [a]
rev []      = []
rev (y:ys)  = (rev ys) ++ [y]

myhead :: [a] -> a
myhead      = head . (foldr f [])

mylast :: [a] -> a
mylast      = head . rev . (foldr f [])

inc :: Int -> Int
inc x = x + 1

main        = print (   (myhead . map inc) [1..1000]
                      + (mylast . map inc) [1001..1010])
```

The above example program was translated into FLIC and was then executed and profiled using the UCL lexical profiler. The `print` function was dropped from the FLIC version (our experimental reducer does not yet offer a `[Response]`− > `[Request]` interface) and `head` and `map` were defined in the standard prelude.

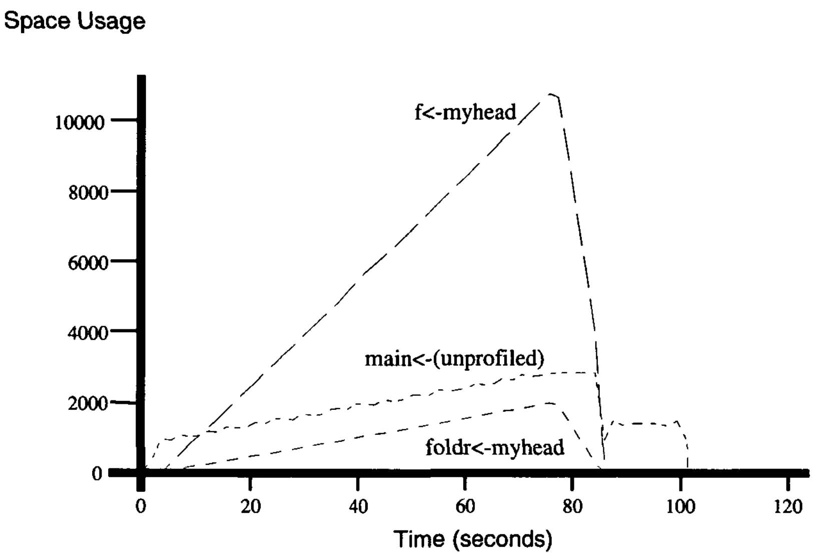

Fig. 21. A space-profile of a blocked pipeline.

The following functions were profiled: `f, rev, myhead, mylast, head, foldr,` and `main`. Both `map` and `inc` were subsumed by the profile for `main`.

The example program ran in just over 111 seconds, with the profiler reporting 23.36 seconds for garbage collection. The call-count results shown in Fig. 20 demonstrate that 1000 calls were made to `f` from `myhead` and 10 calls were made to `f` from `mylast`: the call-count for the former is suspiciously high. Furthermore, the space-profile given in Fig. 21 shows that the majority of heap allocations are made by `f` when called from `myhead`, even though we might reasonably expect `myhead` to extract the first item without constructing the entire list (for clarity, the space profile only shows results for `f<—myhead`, `main<—(unprofiled)` and `foldr<—myhead`: all other results were insignificant).

Here, `f` (needlessly) pattern matches on its second argument and therefore requires `foldr` to have built the tail of the list before it can return its result. The partial application `foldr f []` is therefore tail-strict, which entails unnecessary work for `myhead`; this is an extreme example of a pipeline blockage.

The space-profile demonstrates the user-friendliness of lexical profiling. The lexical profiler reports that the faulty function is `f` *as used in the definition of* `myhead`, whereas a dynamic profiler would report that the faulty function was `f` *called from* `foldr`. Thus, a dynamic profiler would leave the programmer in some doubt as to whether the problem occurs in `myhead` or `mylast`. In a large program, `foldr` might be used in a hundred different places: by contrast, the definition for `myhead` only appears once.

The program may be modified to remove the unnecessary pattern matching from `f`:

```
f :: a -> [a] -> [a]
f x y = x:y
```

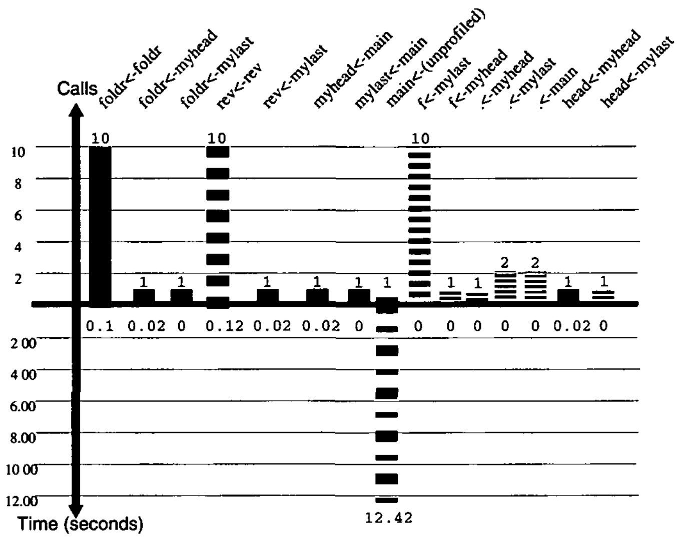

Fig. 22. Call counts and time profile for an unblocked pipeline.

The results of profiling the new program (given in Fig. 22) show that only one call is now made to `f` from `myhead`. The program took just over 16 seconds to run, including 3.76 seconds of garbage collection.

The new space-profile is given in Fig. 23: notice that the scales used for both axes are very different from those used in the initial space-profile (and again notice that we have only labelled the significant data). The space-profile shows that the pipeline blockage has successfully been eliminated: the peak space usage has been reduced from over 10,000 cells to about 1,400 cells, and the execution time has dropped from 111 seconds to about 16 seconds. Furthermore, by comparing the shapes of the curves we observe that all of the work performed by `main` in Fig. 21 during the first 85 seconds seems to have been eliminated.

7.2 Example profile of the nqueens program

This program tries to put *n* queens on a chess board such that they are all safe. The program can try placing from 1 queen up to 8 queens on the board and returns a list of all the valid results. In the following test program the first 10 elements, with 7 queens on the board, are calculated:

```
queens :: Int -> [[Int]]
queens 0     = [[]]
queens (m+1) = [ p++[n] | p<-queens m, n<-[1..8], safe p n ]

safe :: [Int] -> Int -> Bool
safe p n = all not [ check (i,j) (m,n) | (i,j) <- zip [1..] p ]
           where m = 1 + length p
```

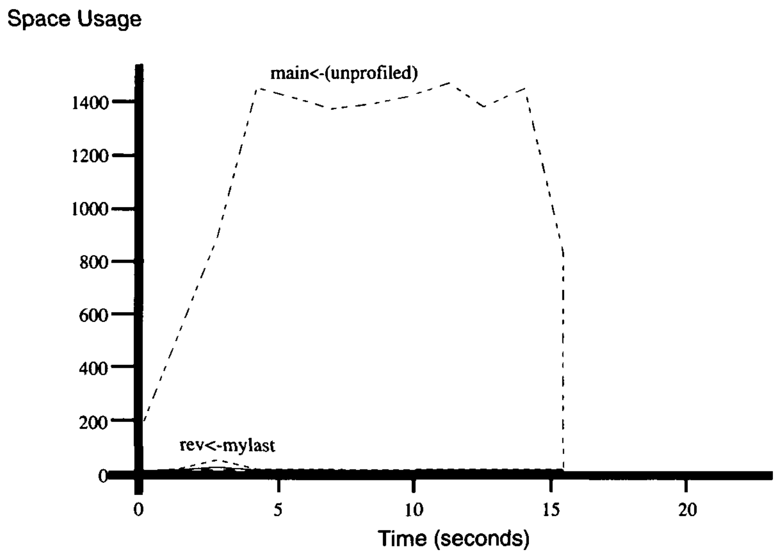

Fig. 23. A space-profile after the pipeline is unblocked.

```
check :: (Int,Int) -> (Int,Int) -> Bool
check (i,j) (m,n)
    = j==n || (i+j==m+n) || (i-j==m-n)

all :: (a -> Bool) -> [a] -> Bool
all p = and . map p

and :: [Bool] -> Bool
and = foldr (&&) True

main = take 10 (queens 7)
```

The program was compiled for profiling, and results were requested for the functions `queens`, `safe`, `check`, and `main`. Some of the other functions were shared and were therefore automatically profiled: however, the shared functions made an insignificant contribution to the computation and in the interests of clarity we have therefore not presented this data. The call-count and timing data for this program are given in Fig. 25 and the space-profile is given in Fig. 24: the clarity of both figures has been improved by discarding insignificant data.

The space results presented give similar information to the Runciman and Wakeling heap profiler but are in a different form. Runciman and Wakeling present their data as cumulative strata, whereas the lexical profiler presents the data for each function absolutely[††].

Attention is drawn to the `queens` function as it uses the most space. The lexical profiler also produces call-count and timing data. By analysing Fig. 25, attention is

[††] It would not be difficult to post-process the space usage data to generate a report in the style of Runciman and Wakeling.

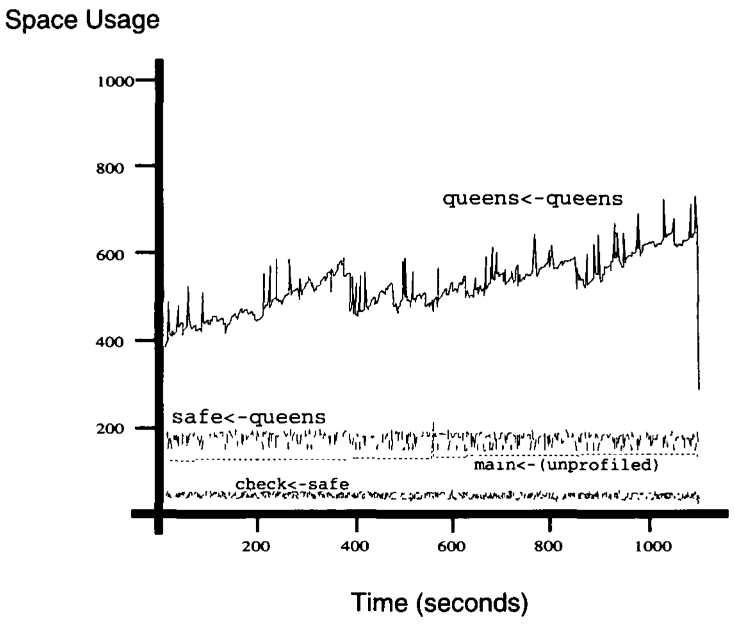

Fig. 24. Heap usage results for nqueens program

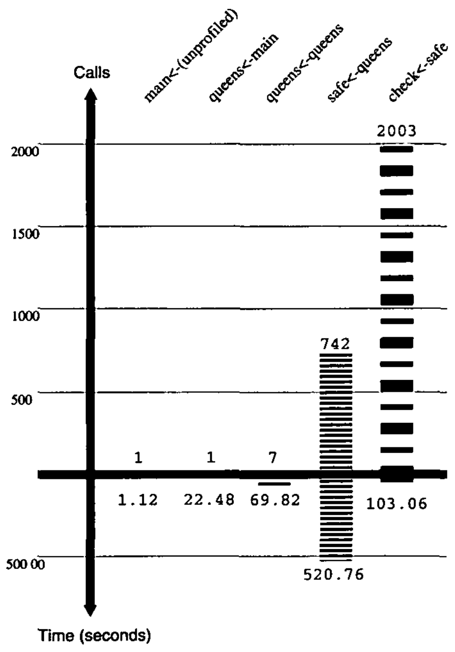

Fig. 25. Call-count and timing results for nqueens.

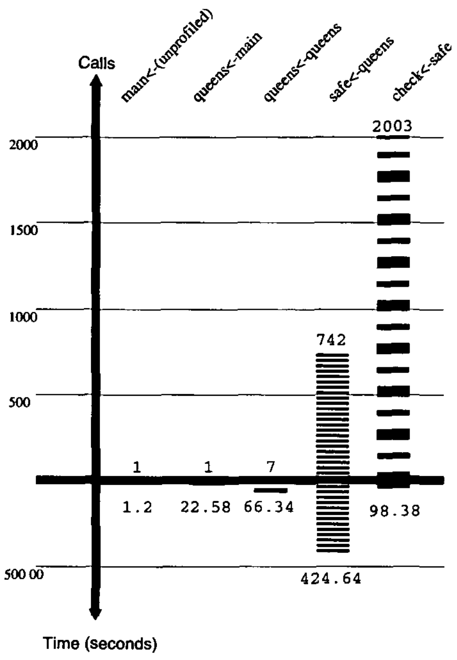

Fig. 26. Call-count and timing results for new nqueens.

drawn to the function safe. It has been called on 742 occasions with an accumulated time of 520 seconds (70% of the program execution time). Clearly, the safe function might benefit from some optimization. The body of safe is primarily an application of all to the function not and a list comprehension. Since the first argument is a compile-time constant, the partial application all not can be redefined as:

```
allFalse :: [Bool] -> Bool
allFalse []         = True
allFalse (True:r)   = False
allFalse (False:r)  = allFalse r
```

and safe can be redefined as:

```
safe p n = allFalse [ check (i,j) (m,n) | (i,j) <- zip [1..] p ]
           where m = 1 + length p
```

The call-count and time data of the new version of the program are presented in Fig. 26 (the new space profile is not significantly different from Fig. 24 and so is not shown).

The function safe now executes in 80% of the time that it used to and the whole program is 15% faster. This has demonstrated the benefit of profiling call-counts and time in addition to space.

```
sumSquares :: Int -> Int
sumSquares n   =  (sum . map square . upto 1) n

upto :: Int -> Int -> [Int]
upto n m       =  if n > m then []
                  else n : upto (n+1) m

square :: Int -> Int
square x       =  x*x

main           =  sumSquares 400
```

Fig. 27. The sum of squares program.

7.3 Verifying program behaviour – sum of squares

In this section the profiler is used to verify whether a hand-coded function performs better than a function-composition pipeline which does the same job.

This example is a program to sum the squares of a list of numbers. Ferguson and Wadler (1988) suggest that the pipelining style of programming (through the use of function composition), which is common in functional languages, is inefficient as there is a need to build and immediately destroy intermediate list elements. A more efficient version can be written which has the same semantics and operational behaviour as the pipelining version. However, this efficient version has the disadvantage that it is considerably less clear than the pipelining version. In this section, the profiler is used to verify Ferguson's statement. Ferguson defines the sum of the squares to be:

```
(sum . map square . upto 1) n
```

A program to evaluate this expression is shown in Fig. 27. By profiling this program, the results obtained for call-count and function times are displayed in Fig. 28 and the heap usage results are displayed in Fig. 29.

It is interesting to note that the precise quantitative data provided by the call-count profile can be just as useful as the qualitative comparison of the relative heights of columns on the graph. For example, the first call-count profile of our `sumSquares` program was rather surprising in that it indicated 399 recursive calls to `upto`, rather than the expected 400. Thus, a difference of just 1 call highlighted a bug in the program: the use of $>=$ instead of $>$ in the definition of `upto`. The utility of call-counting has a long history: the first imperative profiles were mainly concerned with call counts for performance and debugging, as illustrated in Knuth (1971).

A second version of the sum of squares program, which Ferguson says is more efficient, is given in Fig. 30. The results of profiling this program are displayed in Fig. 31 and Fig. 32.

The profiling results demonstrate that Ferguson is correct in stating that the second version of sum of the squares is faster, because the second version executes in 14

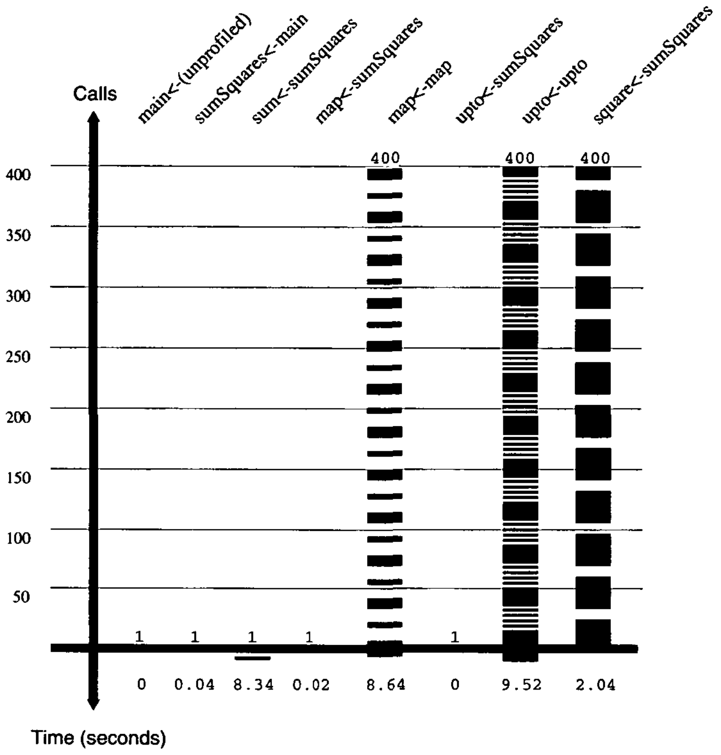

Fig. 28. Call-count and timing results for sumSquares

seconds whereas the first version took 36 seconds. However, in the second version, the space usage has a larger peak than the first version. Thus, if space utilization is more important for an application than speed of execution, the function-composition style is preferable. The subtle effects of laziness mean that such trade-offs are not always obvious to the programmer and therefore a profiler which offers both time and space measurement can be of great assistance.

8 Summary and current status

One of the major problems in developing applications in lazy, functional languages is the lack of tools which aid the programmer in debugging and analysing the run-time behaviour of the application. This problem is compounded by the fact that functional programmers and functional language implementors have different requirements and view program behaviour in different ways.

We have examined the different perspectives of programmers and implementors and have examined the different existing profiling techniques. Most existing profilers are either limited in the information they provide, or only provide information in a way that is useful for language implementors, or both. A notable recent success

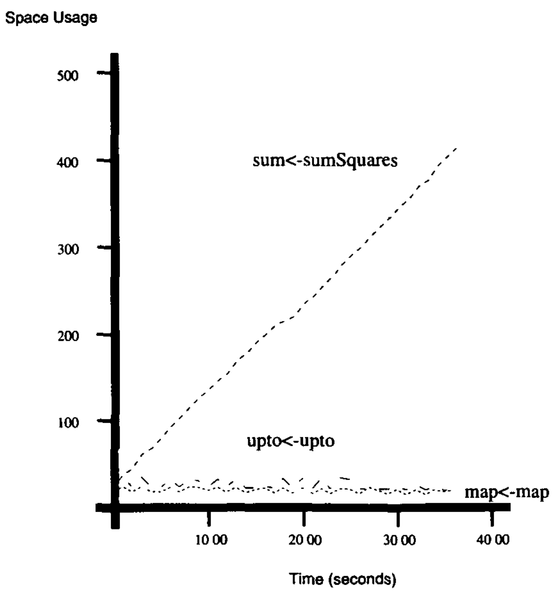

Fig. 29. Heap usage results for sumSquares program

```
sumNsquares n           =  sumNsquares' 0 1 n

sumNsquares' res m n    =  if m > n then a
                           else
                           sumNsquares' (res + square m) (m+1) n

main                    =  sumNsquares 400
```

Fig. 30. The alternative sum of squares program.

has been the heap profiler from Runciman and Wakeling (1993). By contrast, we have concentrated on providing a profiler which will (a) provide time, space and call-count profiles, and (b) present information in a way that is more meaningful for applications programmers.

We have presented the design and implementation of a profiler which measures call-count, time, and heap space usage of lazy, higher-order functional languages using a new technique called *lexical profiling.*

Lexical profiling collects information about the run-time behaviour of functional programs, and reports the results with respect to the way programs are written rather than to how they are evaluated. It is important because it provides a view of program activity which is largely independent of the underlying evaluation mechanism, and therefore programmers may easily relate results back to the source program. Furthermore, neither profiling annotations nor primitives need to be learned as lexical profiling allows the program to be executed without alteration.

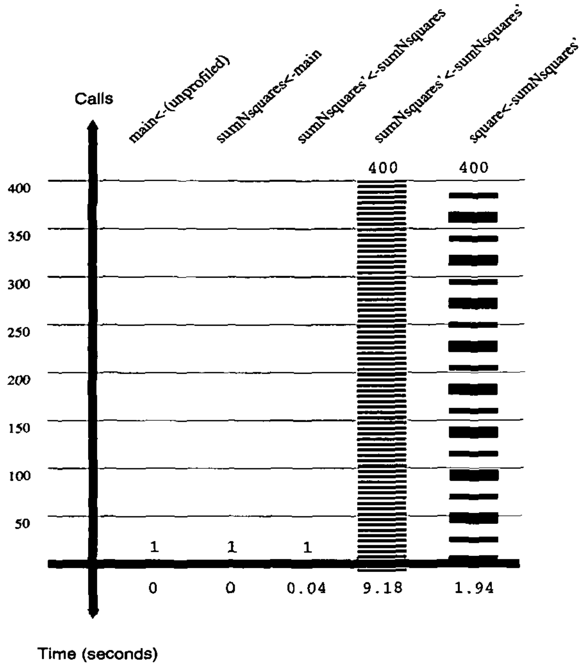

Fig. 31. Call-count and timing results for sumNsquares

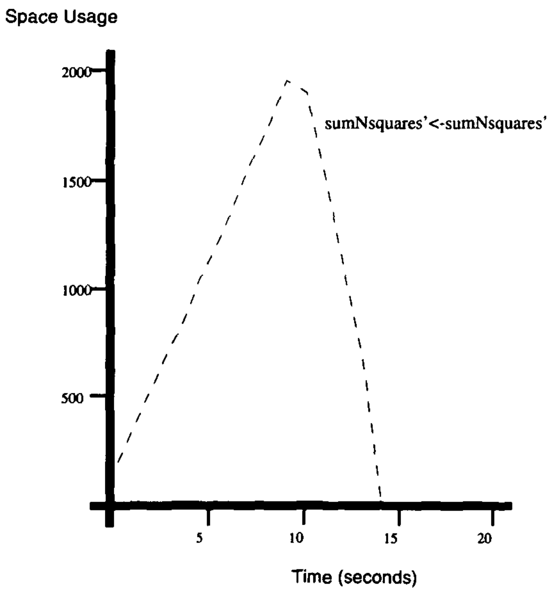

Fig. 32. Heap usage results for the sumNsquares program

Full implementation details have been provided for a sequential, interpretive graph reduction engine, and we have analysed three example programs to show how lexical profiling may be used to detect a pipeline blockage, to illustrate the importance of time profiling and to verify previous claims regarding the use of intermediate lists.

A sequential interpretive profiler has been in use at UCL for several months and has proved invaluable for detecting bugs in Haskell programs (the interpretive reducer is only used for profiling, not for full-blooded evaluation). The current version is not quite complete in that it uses FLIC code (Peyton Jones and Joy, 1989) as its input – we have not yet modified the Haskell compiler, so although we can profile Haskell programs it is necessary to provide function names as seen in the FLIC intermediate code. Our output is not as pretty as that produced by the Runciman and Wakeling heap profiler, but is sufficient for our prototype implementation. A parallel implementation based on the DIGRESS system at Athena Systems Design Ltd. is now well advanced and we hope to start work on a compiled version soon. We have not yet felt the need to implement the statistical style of profiling.

9 Further work

The techniques presented in this paper are illustrated using a sequential, interpreted model of graph reduction. However, they can also be implemented as part of a fully compiled abstract machine and on a parallel reduction architecture. These extensions are discussed below. Additionally, we wish to follow the success of Runciman and Wakeling (1993) by implementing *constructor profiling*. At present we do not do constructor profiling as our system uses FLIC as its input, and any indication of the names of constructors have been stripped by the Haskell compiler: in the FLIC source only `PACK`s are seen.

Our next implementation step is to modify the Haskell compiler so that we can profile Haskell functions rather than FLIC functions. This will raise three issues: (i) how to profile the extra functions which are created by the compiler, for example during lambda-lifting and other optimizations, (ii) how to preserve colours throughout the various transformations which the Haskell compiler applies to the program, and (iii) how to profile separately compiled modules. So far we have avoided all three of these issues because the user sees the FLIC source code and specifies individual FLIC functions to be profiled – including, if necessary, those functions which have been added by the compiler and whose names are visible to the user. When we modify the Haskell compiler we will need to decide what to do with the additional functions (since their names will not be visible to the user) – it seems reasonable to assign to them the colour of the definition from which they were lifted, though it is not yet clear how the costs of Haskell dictionaries (which are introduced as a result of the type hierarchy) should be attributed. The preservation of colours is something which can only be addressed once we know more about the internals of the Haskell compiler. Separate compilation should not present many problems: we expect that compilation of a module to intermediate code will cause all functions to be coloured, and that the linker will resolve the colours to reflect those functions that the programmer wishes to profile.

It has been our intention to provide a statistical profiling mode by post-processing the results of inheritance-mode profiling. This would be fairly straightforward apart from the unresolved issue of what to do with mutually recursive functions. However, we have not yet felt the need for the statistical style and so this extension currently has a low priority.

Finally, we note that lexical profiling provides information about the lexical affiliations of run-time constructs which could be used as the basis for a lexical debugger: for example, not only recording the values of actual arguments each time a specified function is called, but also recording for each call *where* that application occurs in the text of the program.

9.1 Extending lexical profiling to compiled graph reduction

Compiled graph reduction typically makes much more use of the stack for calculations which do not need to be written out to the heap. The heap is used when shared objects such as closures and data structures are built (Fairbairn and Wray, 1987; Burn *et al.*, 1988; Peyton Jones and Salkild, 1989; Augustsson and Johnsson, 1989b). To implement our profiling technique for compiled abstract machines such as TIM (Fairbairn and Wray, 1987) or the Spineless Tagless G-Machine (Peyton Jones and Salkild, 1989), it will therefore be necessary to ensure that space utilization of the stack is monitored, in addition to space utilization of the heap. Time profiling and call-count profiling will be similar to the interpretive implementation. However, template instantiation in an interpretive reducer is replaced by a code sequence in a compiled reducer. Thus, it will be necessary to modify the code generator to produce code to deal correctly with colour manipulation. We have already started work on a lexically-profiled, lazy version of the TIM compiled graph reducer (Fairbairn and Wray, 1987, p. 38): we will report our results in a subsequent paper (Clack and Parrott, 1993).

9.2 Extending the technique to parallel graph reduction

Lexical profiling can be extended to parallel graph reduction by distributing the *profile tables* among the processing elements of the parallel machine.

For post mortem profile reports, each processing element will have a local *profile table* which will be updated in the usual manner. At the end of a program run, a global *profile table* is generated from the collected local tables. Once a program is coloured it is not significant where a function executes.

A parallel profiler is currently being implemented on the DIGRESS system at Athena Systems Design Ltd. This profiler not only provides post-mortem profile results, but also provides a dynamically updateable global *profile table* which can be used to report heap usage as the program executes ('on-the-fly' profiling helps to debug programs which run but do not produce output within a reasonable time – this common problem is often due to to an overly strict function in a pipeline which may attempt an infinite evaluation, and in a large program a profiler is necessary in order to detect *which* pipeline is defective). This requires a special purpose processing element dedicated to the collection and presentation of data. The other processing elements are responsible for supplying incremental profiling information at regular intervals. The key challenge here is to strike a balance between keeping displayed information up to date and swamping the parallel computation with profile messages.

Acknowledgements

We are greatly indebted to Rex Page, of the Amoco Production Company's Tulsa Research Center, for his encouragement and support of this work. We also thank Athena Systems Design Ltd. for their help with the parallel implementation of the profiler and Dennis Parrott for debugging the implementation and for proof-reading drafts of this paper. Finally, we thank our two referees for their detailed comments and suggestions which have enhanced the presentation of this material.

References

Appel, A. W., Duba, B. F. and MacQueen, D. B. (1988) Profiling in the presence of optimization and garbage collection. Distributed with the New Jersey SML compiler.

Augustsson, L. and Johnsson, T. (1989a) The Chalmers Lazy ML compiler. *The Computer Journal*, **32**(2): 127–141.

Augustsson, L. and Johnsson, T. (1989b) Parallel graph reduction with the $\langle v, G\rangle$-machine. In: *Proc. FPCA Conference*, pp. 202–213. ACM.

Augustsson, L. (1984) A compiler for Lazy ML. In: *Symposium on Lisp and Functional Programming*, pp. 218–227. ACM.

Axford, T. H. (1990) Reference counting of cyclic graphs for functional programs. *The Computer Journal.*

Baker, H. G. (1978) List processing in real time on a serial computer. *Comm. ACM*, **21**(4): 280–294.

Burn, G. L., Peyton Jones, S. L. and Robson, J. (1988) The Spineless G-Machine. In: *Proc. Lisp and Functional Programming Conference*, pp 244–258. Snowbird, UT.

Burn, G. L. (1987) Evaluation transformers – a model for the parallel evaluation of functional languages. In: *Proc. FPCA Conference*, pp. 446–470. ACM, Springer Verlag. (extended abstract), LNCS 274.

Cohen, J. (1981) Garbage collection of linked data structures. *ACM Comput. Surv.* **13**(3).

Clack, C. D. and Parrott, D. J. (1993) Compiled lexical profiling for tim. In preparation.

Clack, C. D. and Peyton Jones, S. L. (1986) The four-stroke reduction engine. In: *Proc. Lisp and Functional Programming Conference*, pp. 220–232. ACM.

Clayman, S. (1993) *Developing and Measuring a Parallel Rule Based System in a Functional Programming Environment.* PhD thesis. University College London.

Fairbairn, J. and Wray, S. C. (1987) Tim: A simple, lazy abstract machine to execute supercombinators. In: *Proc. FPCA Conference. Lecture Notes in Computer Science Vol. 274.* Springer-Verlag.

Ferguson, A. B. and Wadler, P. (1988) *When will deforestation stop.* Technical report, University of Glasgow, Department of Computing.

Graham, S. L., Kessler, P. B. and McKusick, M. K. (1982) `gprof`: a call graph execution profiler. *ACM Sigplan Notices*, **17**(6): 120–126.

Glaser, H., Reeve, M. and Wright, S. (1988) *An analysis of reference count garbage collection schemes for declarative languages*. Technical report, Imperial College London.

Grant, P. W., Sharp, J. A., Webster, M. F. and Zhang, X. (1993) Some issues in a functional implementation of a finite element algorithm. In: *Proc. FPCA Conference*. ACM.

Hudak, P. (1986) A semantic model of reference counting and its abstraction (detailed summary). In: *Proc. Lisp and Functional Programming Conference*, pp. 351–363. ACM.

Hughes, J. (1985) A distributed garbage collection algorithm. In: *Proc. FPCA Conference*, pp. 256–272. ACM,

Hughes, J. (1987) *Managing reduction graphs with reference counts*. Technical Report CSC/87/R2, University of Glasgow.

Hughes, J. (1989) Why functional programming matters. *The Computer Journal*, **32**(2): 98–107.

Johnsson, T. (1984) Efficient compilation of lazy evaluation. In: *Proc. Conference on Compiler Construction*, pp. 58–69. ACM.

King, I. (1990) The efficiency and generalisation of the various abstract machines. In: M. J. Plasmeijer (ed), *2nd International Workshop on Implementation of Functional Languages on Parallel Architectures*, pp. 255–280. University of Nijmegen.

Kozato, Y. and Otto, P. (1993) Benchmarking real-life image processing programs in lazy functional languages. In: *Proc. FPCA Conference*. ACM.

Knuth, D.E. (1971) An Empirical Study of FORTRAN Programs. *Software – Practice and Experience*, **1**: 105–133.

Lermen, C-W. and Maurer, D. (1986) A protocol for distributed reference counting. In: *Proc. Lisp and Functional Programming Conference*, pp. 343–350. ACM.

Nilsson, H. and Fritzson, P. (1992) Algorithmic debugging for lazy functional languages. In: *Proc. Fourth International Symposium on Programming Language Implementation and Logic Programming*, Linköping, Sweden.

Parrott, D. J. (1993) *Synthesising Parallel Functional Programs to Improve Dynamic Scheduling*. PhD thesis, University College London.

Parrott, D. J. and Clayman, S. (1990) Report on 'cost' and 'debug' primitive extensions to FLIC. Research note RN/91/79, Department of Computer Science, University College London.

Parrott, D. J. and Clack, C. D. (1991) A common graphical form. In: *Proc. Phoenix Seminar & Workshop on Declarative Programming*, pp. 224–238. Springer-Verlag. (Also Research Note RN/91/27 Dept. of Computer Science, University College London.)

Parrott, D. J. and Clack, C. D. (1992) Paragon – a language for modelling lazy, functional workloads on distributed processors. In: *Proc. UK Performance Engineering Workshop*. (Also UCL Research note RN/92/72.)

Peyton Jones, S. L. (1987a) The tag is dead – long live the packet. Posting on fp electronic mailing list.

Peyton Jones, S. L. (1987b) *The Implementation of Functional Programming Languages*. Prentice Hall.

Peyton Jones, S. L. and Joy, M. S. (1989) FLIC – a Functional Language Intermediate Code. Internal Note 2048, University College London, Department of Computer Science.

Peyton Jones, S. L. and Lester, D. R. (1992) *Implementing Functional Languages: a tutorial*. Prentice Hall.

Peyton Jones, S. L. and Salkild, J. (1989) The Spineless Tagless G-Machine. In: *Proc. FPCA Conference*, pp. 184–201.

Rudalics, M. (1986) Distributed copying garbage collection. In: *Proc. Lisp and Functional Programming Conference*, pp. 364–372. ACM.

Runciman, C. and Wakeling, D. (1992) Heap profiling of a lazy functional compiler. In: *Proc. Glasgow Workshop in Functional Programming*. Springer-Verlag.

Runciman, C. and Wakeling, D. (1993) Heap profiling of lazy functional programs. *J. Functional Programming*, 3(2).

Shute, M. J. (1988) *Y-less execution through fractional reference-counting*. Draft report, Middlesex Polytechnic.

Sansom, P. M. and Peyton Jones, S. L. (1992) Profiling lazy functional languages. In: *Proc. Glasgow Workshop in Functional Programming*, pp. 227–239. Springer-Verlag.

Sansom, P. M. (1993) Time profiling a lazy functional compiler. In: *Proc. Glasgow Workshop in Functional Programming*. Springer-Verlag.

Sansom, P. M. (1994) *Execution Profiling for Non-strict Functional Languages*. PhD Thesis (in preparation), University of Glasgow, Department of Computing.

Tolmach, A. P. and Dingle, A. T. (1990) Debugging in Standard ML of New Jersey. Distributed with the New Jersey SML compiler.

Turner, D. A. (1985) Miranda: A non:strict functional language with polymorphic types. In: *Proc. FPCA Conference*, pp. 1–16. ACM,

Wright, P. J. (1994) *Optimised Redundant Cell Collection for Graph Reduction*. PhD thesis, University College London, Department of Electronic Engineering.

Zorn, A. B. and Hilfinger, P. (1988) A memory allocation profiler for C and LISP programs. In: *Proc. USENIX Conference*, pp. 223–237.