Profiling Lazy Functional Languages
Working Paper

Patrick M Sansom and Simon L Peyton Jones*
University of Glasgow

May 18, 1992

Abstract

Profiling tools, which measure and display the dynamic space and time behaviour of programs, are essential for identifying execution bottlenecks. A variety of such tools exist for conventional languages, but almost none for non-strict functional languages. There is a good reason for this: lazy evaluation means that the program is executed in an order which is not immediately apparent from the source code, so it is difficult to relate dynamically-gathered statistics back to the original source.

We present a new technique which solves this problem. The framework is general enough to profile both space and time behaviour. Better still, it is cheap to implement, and we describe how to do so in the context of the Spineless Tagless G-machine.

1 Introduction

Many functional programs are quick and concise to express (Hughes [1989]) but often slow to run. Before being able to improve the efficiency of a program, a programmer has to be able to:

1. Identify the execution bottlenecks or "critical parts" of the program that account for much of the time and space used. This allows effort spent improving the program to be focussed on parts of the program where it will be of greatest benefit.
2. Identify any inefficiencies present in the bottlenecks thus identified. These may range from "hidden" space leaks (Peyton Jones [1987]), caused by the subtleties of the method of evaluation, to inappropriate choices of algorithms and data structures.

Once this is done alternative, more efficient, solutions can be proposed and evaluated. This "performance debugging" process should be carried out on real programs running with real data, so the true performance can be tuned.

Conventional languages provide profiling tools such as *gproff* (Graham, Kessler & McKusick [1983]) and *mproff* (Zorn & Halfinger [1988]) attribute time usage and space allocation to

*Authors' address: Department of Computing Science, The University, Glasgow, G12 8QQ, UK.
E-mail: `sansom@dcs.glasgow.ac.uk`, `simonpj@dcs.glasgow.ac.uk`

source code. This enables the programmer to identify the “critical parts” of the program being developed. However current functional language programming environments lack equivalent tools. We are interested in developing such tools to assist with profiling the performance of lazy functional programs.

Though these tools will include the identification of *time* spent in the different “parts” of the program, we are also interested in tools which identify the *space* usage. Unlike most conventional languages, functional languages provide an abstraction which hides the allocation and reclamation of data structures. This abstraction can result in unexpected spatial behaviour ranging from over-liberal allocation to so-called *space leaks* (Peyton Jones [1987]). Initial results from Runciman and Wakeling who have developed a heap profiling tool have indicated how revealing such information can be (Runciman & Wakeling [1992]).

An essential part of any profiler is that the profiling information gathered must be *faithful* to normal execution. In particular:

- The lazy evaluation order must not be modified.
- We want to profile the actual events which occur during real compiled execution. An instrumented interpreter is not satisfactory.
- The profiling scheme must be reasonably cheap. The additional cost of profiling should not distort the information gathered about the execution.

Our approach concentrates on what we consider to be the key problem: The identification of the different program “parts” for which statistics are accumulated (Section 2). We introduce the notion of a “cost centre” which identifies the source code expressions for which we are interested in accumulating statistical information (Section 3). We go on to describe a remarkably simple implementation which keeps track of the current source expression being evaluated without modifying the lazy semantics (Section 4) and finally describe how we use this scheme to gather the profiling statistics we are interested in (Section 5).

2 Profiling Lazy Functional Programs is Difficult

The key problem in profiling any program is to relate any information gathered about the execution of the program back to the source code in a way which accurately identifies the source that was responsible it. Unfortunately, the very features which lazy functional languages advocate, such as higher order functions, polymorphism, lazy evaluation and program transformation, make a well-defined mapping back to the source code difficult to achieve.

A general discussion of some of the problems that lazy languages pose to profiling is given in Runciman & Wakeling [1991]. We address the issues with respect to the mapping problem identified above.

2.1 Re-use of Functions

Polymorphism and higher order functions encourage the re-use of functions in many different contexts. In Hughes [1989], the ability to provide generalised, higher order functions which can then be specialised with appropriate base functions is advocated as a major strength of functional languages. Unfortunately this heavy re-use of a small number of functions makes it harder to identify the source of observed execution costs. Consider the expression:

```
map (f x) l
```

Knowing that the program spent 20% of its time in the function `map` is not particularly helpful, as there will probably many applications of `map` in the program. The costs of the different calls to `map` need to be attributed to their call sites. In general we may want to attribute the cost of a function invocation to the *stack* of call sites invoking its execution.

This problem does arise in conventional languages where a routine may be called from many call sites, which are usually statically identifiable. It is normally solved by using an approximation which apportions time spent in a particular routine to its various callers (Graham, Kessler & McKusick [1983]).

2.2 Lazy Evaluation

Lazy evaluation poses the profiler with some awkward problems.

It is not necessarily clear what part of the program should bear the cost of evaluating a suspension. An expression is only evaluated if its result is demanded by some other expression. So the question arises: "Should the cost of evaluation be attributed to the part of the program which instantiated the expression or the part of the program which demanded the result?". This is further complicated by the fact that multiple expressions may demand the result with all but the first finding the expression already evaluated. If we attribute the cost to demanding expressions it should probably be shared among all the demanding expressions.

Furthermore, the nature of lazy evaluation means that evaluation of an expression will be interleaved with the evaluation of the inputs which it demands. As this expression is itself being demanded it will also be interleaved with the execution of its demander. The resulting order of execution bears no resemblance to the source code we are trying to map our profiling results to. A scheme which attributes the different execution fragments to the appropriate source expression is required. Accumulation of statistics to the different call sites is made more difficult as we do not have an explicit call stack — instead we have a demand stack.

It is also very desirable that the lazy semantics are not modified by the profiler. In conventional languages one might measure the time taken to execute between two "points" in the source. However in a lazy language there is no linear evaluation sequence so we no longer have the notion of a "point" in the execution corresponding to a "point" in the source. One could imagine a crude profiling scheme which forced evaluation of the intermediate data structure after each phase of (say) a compiler. This would enable us to measure the cost of the each phase, but we would be measuring the cost of a different program — one which forces its intermediate data and which may be evaluating parts which need never be evaluated!

2.3 Program Transformation and Optimisation

Functional language implementations involve radical transformation and optimisation which may result in executable code which is very different from the source:

- Hidden functions are introduced by high level translation of syntactic sugar such as list comprehensions.
- Auxiliary functions and definitions are introduced as expressions are transformed.
- The combined effect of all the transformations may drastically change the structure of the original source.

It is highly undesirable to turn off these optimisations as the resulting profile would not be of the program you actually want to run and improve. As we want to be able to profile a fully optimised program, running with real data, this transformation problem must be addressed.

3 Cost centres — A practical solution

Our approach specifically addresses the problem of mapping the statistics to source code. Rather than collecting profiling information about every function in the source, we annotate particular source expressions, which we are interested in profiling, with *cost centres* which identify the expressions. During execution statistical information is gathered about the expressions being evaluated and attributed to the appropriate cost centre.

Suppose we are interested in determining the cost incurred in evaluating the expression

```
map (f x) l
```

We introduce a new expression construct `scc` which is used to annotate an expression with a cost centre. A *cost centre* is a label under which we attribute execution costs. It is represented as a literal string which identifies the cost centre. For example

```
scc "foo" (map (f x) l)
```

causes the cost of the actual evaluation of `map (f x) l` to be attributed to the cost centre `"foo"`. This expression will be used as a running example throughout this section.

The `scc` expression construct encloses a static expression whose evaluation cost is associated with a cost centre identified by a *label*

`scc` *label expr*

We insist that the label is a literal string as we do not want to have to compute it at runtime. Semantically, `scc` simply returns the value of *expr*, but operationally, it results in the cost of evaluating *expr* to be attributed to the cost centre *label*.

For the collected statistics to have any useful meaning we must clearly identify what we mean by the *cost incurred in evaluating an expression*. This requires us to define exactly what evaluation should be associated with a particular source expression. The following questions must be addressed:

1. How far is the expression evaluated?
2. Where is the evaluation of unevaluated arguments attributed?
3. What happens when cost centres are nested?
4. What about multiple instances of the expression being evaluated?

These questions must be answered in a way which is comprehensible to the programmer. That is, independent of any operational matters such as the order of evaluation.

3.1 Degree of evaluation

We are measuring the cost of a source expression. However in a lazy language the extent to which an expression is evaluated is dependent on the demand placed by the surrounding context.

For our running example there are numerous possibilities, some of which are given below:

- The list and its elements are completely evaluated, as in

```
let  foo = scc "foo" (map (f x) l)  in  foldr (+) 0 foo
```

- The spine is evaluated, but the elements are left unevaluated, as in

```
let  foo = scc "foo" (map (f x) l)  in  length foo
```

- The list is not evaluated at all.

We don't want to affect the lazy evaluation sequence at all — we want to measure the evaluation that is actually required by the program being executed and at the same point in time that it is required. Let's call this degree of evaluation the *actual evaluation*.

This unknown degree of evaluation results in a potential source of confusion. For example, the programmer might be expecting to measure evaluation which never occurs. However, we are interested in identifying the critical expressions within the program, and are not concerned with potentially inefficient expressions which are never actually evaluated. If the evaluation is demanded it will be measured.

3.2 Unevaluated arguments

Should the evaluation cost of unevaluated arguments be attributed to the demanding expression or the expression which created the unevaluated closures?

When examining the cost of a particular expression we don't want the water to be muddied by the degree of evaluation of the expression's inputs, we just want to know the cost of execution the expression itself. This corresponds to the intuition we have from strict languages where the evaluation of the inputs does not affect the evaluation of an expression as it will already have occurred.

So we do *not* want the cost of evaluating `x` and `l` to be attributed to the cost centre `"foo"` — they are attributed to the cost centre which was responsible for building them. What we do want to know is the cost incurred executing this particular call of `map` and all the calls to `f` hidden inside `map`. Calls to `map` from other sites will be attributed to the enclosing cost centre at that site.

3.3 Nested Cost Centres

It is possible for an expression enclosed with an `scc` annotation to have an `scc` expression within it. Consider the expression

```
scc "foo" map (f x) (scc "bar" (map (g y) l))
```

Should the cost of the inner `map`, be attributed to the cost centre `"foo"` as well as `"bar"`? We adopt a very simple scheme. Inner sub-expressions wrapped in an `scc` annotation will be attributed to the cost centre specified there. That is, *costs are only attributed to a single cost centre.* So the costs of the inner `map`, `map (g y) l`, will be attributed to the cost centre `"bar"` and the cost of mapping `f x` over the result attributed to `"foo"`.

As another example consider

```
scc "foo" (map (scc "bar" (f x)) l)
```

This will attribute the cost of evaluating `f x` to `"bar"`. The cost of applying the resulting function to the elements of the list `l` will still be attributed to `"foo"` as this is not in the scope of the `scc` for `"bar"`. If we wanted to attribute the cost of evaluating the list elements to `"bar"` we would have to bring the complete application of `f` into the scope of `"bar"`. This can be done by introducing a lambda

```
scc "foo" (map  (\y -> scc "bar" (f x y))  l)
```

Here the cost of the evaluation of elements of the resulting list are attributed to the cost centre `"bar"` while the cost of constructing the list is attributed to the cost centre `"foo"`. If the list was forced but the elements never evaluated the cost of building the list would accrue to `"foo"` but no costs would accrue to `"bar"`. This level control of the attribution of costs allows us to break down the detailed costs of evaluation. However care is needed to ensure that the `scc` expressions measure what we intend.

Aggregation of costs to enclosing cost centres is discussed in section 4.2.

3.4 Evaluation of many instances

Many instances of the source expression, with different inputs may be evaluated during the execution of the program. We do not want to distinguish between the different instances — the source expression is responsible for them all. So we associate with the expression the cost of the *actual evaluation*, ignoring the evaluation of the inputs, of all these instances.

A count of the number of instances involved may be kept so that we can average the costs over the instances. This average may be distorted as the demand on the different instances may be quite varied.

3.5 Identifying Expressions

Initially we plan to leave the identification of source expressions annotated with cost centres to the programmer. This interface decision is independent of the mechanism described which gathers the statistics. Possible alternatives are:

Automated Cost Centres would require the compiler to identify the expressions which will be annotated with cost centres. This would avoid any explicit source modification by the user. Examples might include

- Annotating each "significant" function with a cost centre of the same name.
- Annotating each module by annotating each exported function in the module with the module name.

Optional Cost Centres. An automated scheme could result in a prolific number of cost centres. This would allow the user to decide which cost centres are to be active during a particular run — thus reducing the "noise" associated with parts of the program that they are not interested in.

4 Implementation

The main idea is to keep track of the cost centre of the current expression which is being evaluated — the *current cost centre*. As costs are incurred they are attributed to this current cost centre.

We are implementing cost centres within the framework of the Haskell compiler being developed at the University of Glasgow. This uses the STG-machine (Peyton Jones [1992]) as an underlying abstract machine. The discussion which follows attempts to provide a general description of the manipulation of cost centres. However some details of the STG-machine are revealed to provide a more concrete discussion. The cost centre manipulation is made precise by extending the operational semantics of the STG-machine to include cost centres. These extensions are given in appendix A.

During compilation we statically declare a *cost centre* structure for each `scc` label encountered in the source. `scc` constructs which have the same label refer to a single cost centre. A pointer to this structure is used to identify the cost centre. A store location, `currentCostCentre` is declared, which indicates where costs should be attributed. This is actually a pointer to the appropriate structure.

The main problem with keeping track of the current cost centre in a lazy language is the interleaving of evaluation. When an unevaluated closure, or *thunk*, is subsequently entered we must ensure that

1. The costs of evaluation are attributed to the appropriate cost centre.
2. Once evaluation is complete the cost centre of the demanding expression is restored.

Consider the expression

```
scc "foo" (g x, y)
```

The pair will be returned with the computation for `g x` suspended within. We must ensure that the evaluation of `g x`, if it is demanded, is attributed to the cost centre `"foo"`. To enable us to do this the `currentCostCentre`, `"foo"`, is attached to the thunk, `g x`, when it is built. On entering a thunk the `currentCostCentre` is set to the cost centre stored in the thunk and the subsequent costs attributed to the thunk's cost centre. In fact, we attach the current cost centre to all heap-allocated closures as this allows us to identify which cost centres were responsible for building the various heap objects.

All that remains to be done is to devise a scheme which restores the demanding cost centre once evaluation of the thunk, `g x`, is complete. Lazy evaluation requires the thunk to be updated with its head normal once evaluation is complete, thus avoiding repeated evaluation. It is precisely at this point which we must restore the demanding cost centre — evaluation of the thunk is complete and control is returning to the demanding expression.

In the STG-machine an update is triggered by an update frame. This is pushed onto the stack when the thunk is entered and removed when the update is performed. This provides us with a convenient place to save and restore the demanding cost centre. We augment the update frame with the demanding cost centre. On entering a thunk

1. An update frame is pushed (as usual) which contains the `currentCostCentre` i.e. the cost centre of the demanding expression.
2. The `currentCostCentre` is then set to the cost centre stored in the thunk being entered.

When the evaluation is complete an update is triggered by the update frame. As well as performing the update the `currentCostCentre` is restored.

For closures which are already evaluated the result is returned as normal. No manipulation of cost centres is required. The small cost of entry and return is attributed to the demanding cost centre.

In this way, costs accrue to the cost centre of the *builder* of a closure, and the *enterer*'s cost centre is restored when evaluation of the closure is complete. At this point the closure is in HNF, but any unevaluated closures lying inside the returned value have recorded inside them the appropriate cost centre.

The only other time the `currentCostCentre` is changed is when an `scc` expression is evaluated. Again we have to save the current cost centre and restore it when evaluation is complete. We already have a scheme for doing this which saves the current cost centre in an update frame. The same idea is used here except that we don't push a real update frame. Instead we push a frame that appears to be an update frame but does not contain a closure to be updated. When triggered it simply restores the saved cost centre. Evaluating an `scc` expression requires

1. A “restore cost centre frame” containing the `currentCostCentre` to be pushed. This appears to be an update frame but does not contain a closure to update.
2. The `currentCostCentre` to be set to the new cost centre associated with the `scc` expression.

Thunks built within the `scc` expression will have this new cost centre attached so costs incured during evaluation of these closures will accrue to this centre. When evaluation of the `scc` expression is complete, the apparent update is triggered which simply restores the saved cost centre.

The sole purpose of the cost centre concept is to provide a user-definable mapping from the profiling statistics back to the source code. Time and space allocation can be attributed to the currently active cost centre using the usual clock/allocation interrupt, which samples the `currentCostCentre` register. The cost centre attached to closures indicates which cost centre was responsible for building the closure. This allows us to profile the heap attributing the heap usage to the cost centres that were responsible for creating it.

4.1 Optimisation Revisited

It is important to avoid program transformations that change the scope of the `scc` expression, as this will change the expression which is evaluated under that cost centre. Consider the following transformation:

$$\ldots\ \texttt{scc}\ cc\ (\ldots e_{sub} \ldots)\ \ldots \quad \Longrightarrow \quad \texttt{let}\ v\ \texttt{=}\ e_{sub}\ \texttt{in}\ \ldots\ \texttt{scc}\ cc\ (\ldots v \ldots)\ \ldots$$

This transformation doesn’t change the meaning of the expression. However, as the expression, e_{sub}, is lifted outside the `scc` expression it will result in an evaluation of e_{sub} no longer being attributed to the cost centre cc. In short, program transformation can move costs from one cost centre to another.

Such transformations must be avoided as we require the execution costs to be accurately mapped back to the appropriate source code cost centre. This conflicts with our goal of measuring the costs of a fully optimised implementation. We still perform the program transformations on the expressions within an `scc` expression and on expressions containing `scc` expressions. What must be avoided is performing a transformation which moves computation from the scope of one cost centre to another. That is, we must not

1. Lift a sub-expression out of an `scc` expression (as above).
2. Unfold a definition which was declared outside the `scc` expression. For example

$$\texttt{let}\ v\ \texttt{=}\ e_{defn}\ \texttt{in}\ \ldots\ \texttt{scc}\ cc\ (\ldots v \ldots)\ \ldots \quad \Longrightarrow \quad \ldots\ \texttt{scc}\ cc\ (\ldots e_{defn} \ldots)\ \ldots$$

It is possible to relax these transformation restrictions provided care is taken to preserve the appropriate cost centres. Sub-expressions can be lifted out of an `scc` expression if they carry the `scc` with them. The lifting example above becomes

$$\ldots\ \texttt{scc}\ cc\ (\ldots e_{sub} \ldots)\ \ldots \quad \Longrightarrow \quad \texttt{let}\ v\ \texttt{=}\ \texttt{scc}\ cc\ e_{sub}\ \texttt{in}\ \ldots\ \texttt{scc}\ cc\ (\ldots v \ldots)\ \ldots$$

Definitions may be unfolded if their cost centre can be determined and the unfolded expression is annotated with this cost centre. We can also allow the unfolding manifest functions as these do not involve any arbitrary computation — they only result in the duplication of code. Thus the in-lining of function definitions is not hindered. We still restrict the unfolding of constructor applications, which are also in head normal form, as this will result in the heap allocation of the constructor closure being moved.

In spite of the restrictions the measured cost of evaluating an identified expression will reflect the true cost of the fully-optimised evaluation of that expression. However the cost of evaluating the program as a whole may be affected by optimisations which were curtailed by the profiling requirements to identify the individual expressions of interest.

4.2 Aggregating Cost Information

One possible shortcoming of the scheme outlined above is that it does not aggregate information about the evaluation of an scc expression to the enclosing expression. In the expression

```
let sub = \z -> scc "bar" ... z ...  in  ... scc "foo" ... (sub y) ...
```

the cost of evaluation sub is attributed to the cost centre "bar". It is not apparent at the call sight of sub that its cost will be attributed to "bar", and not to "foo", as the scc expression is hidden within the definition of sub. If the cost of the call to sub was attributed to "foo" as well as "bar" this potential ambiguity might be avoided. Thus we might consider a scheme where the cost is attributed to all the stack of scc sites responsible for its evaluation.

Traditional profiling systems attribute runtime information to each procedure (or function) declared in the program. These may also be aggregated up the call graph so that the costs incurred by lower level procedures are attributed to their calling procedures. *Gproff* uses an approximate technique which assumes that the expected cost of a procedure called from different call sites is equal. It counts the number of times each call site calls a procedure and, on completion, apportions out the cost of the procedure to the various call sites.

If we assume that all invocations of an scc expression from different cost centre call sites have the same average costs. Counting the number of times a particular cost centre is invoked by another cost centre would enable us to proportion the cost out later. The accuracy of this scheme depends on the validity of the averaging assumption If a particular call site has a cost which consistently varies from the average, any proportioning scheme will distort the results. There are a number of potential sources of variation in lazy functional languages:

- Different call sites may pass arguments which lead to different amounts of computation. In conventional languages this problem tends to be limited to different values or data structures. However the use of higher order functions enables very large variations to be easily introduced by a call site.

- The actual evaluation required depends on the demand placed on the result by the call site. This may very considerably between call sites.

An alternative scheme is to define the currentCostCentre to be the call stack of cost centres and do the aggregation accurately at runtime. Each time we set a new cost centre we push it onto the stack. When a monitored event occurs it must be attributed to *all* the cost centres on the `currentCostCentre` stack. Unfortunately many such stacks will need to be maintained, attached to the various unevaluated computations, since lazy evaluation will cause execution involving a particular stack to be suspended and resumed later. All this would impose a very significant overhead, possibly distorting the execution statistics.

These ideas for aggregation assume that the additional information provided is going to give a significant benefit. However it is not clear that this is the case.

1. If the attribution of profiling information was fixed to a particular language construct, such as the function definition, an aggregation scheme would be of great benefit when trying to evaluate the performance of a large nest of function calls.

   Our cost centres are defined by the user, independently of any other language features, so the user can define them to collect statistics about the exact expressions they are interested in. In particular a nest of function calls can be profiled by simply placing a cost centre annotation at the outermost level. The only potential source of confusion is another cost centre annotation lurking within this nest that the user is unaware of. This may be a problem if it happens to be in another module.

2. Conceptually it is much simpler to think of costs being attributed to one and only one cost centre. It is easier to measure particular costs, such as the evaluation of the spine of a list, if sub-expression costs can be attributed elsewhere.

3. Functional languages encourage the use of recursive and mutually recursive declarations which give rise to cyclic dependencies. In the approximate scheme these must be detected and collapsed to a single point. The stack scheme must avoid pushing a cost centre onto the stack if it is already present otherwise costs will be attributed multiple times to the same cost centre.

It is not clear that this aggregation, given our notion of a user-definable cost centre, is going to provide us with a significant improvement in the profiling information collected. For the time being we do not plan to implement aggregation but experience may cause us to revise this decision.

5 Gathering Profiling Statistics

Having described the mechanism by which we map the profiling statistics back to the source code we now move on to describe the statistics we actually gather. It is important to remember that any metering we do to gather the statistics should not distort the characteristics of normal execution — as it is normal execution that we are attempting to profile. Any additional actions required should be very simple and within memory. They should certainly not require any I/O.

5.1 Information Available

We first identify the information which is immediately available to us.

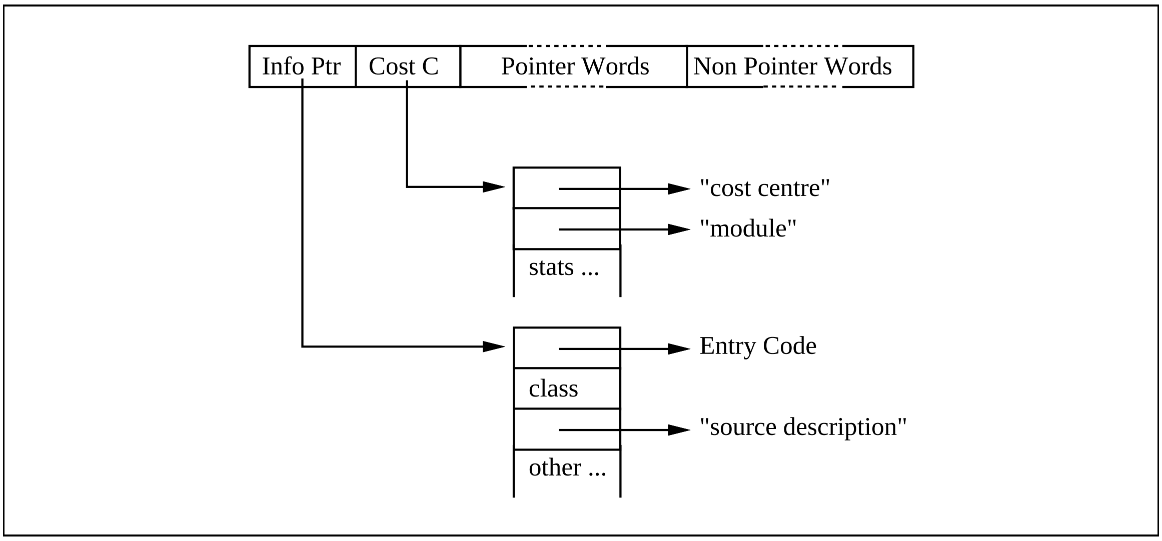

Figure 1: Closure layout

A cost centre is represented as a pointer to a static structure. It consists of:

- The name of the cost centre given with the `scc` annotation.
- The module and line number where the `scc` annotation occurred.
- Any statistical meters we want to accumulate for the cost centre.

At every point during the execution we can identify the current cost centre. This is used to attribute execution events such as a timer interrupt or heap allocation.

Closures also contain information which can be used to profile the heap. In the STG machine, augmented with our implementation of cost-centres, each closure occupies a contiguous block of words, laid out as shown in figure 1. The first word of the closure is called the *info pointer*, and points to the *info table*. The info table is static and contains information about the closure. In particular it contains the code to execute when the closure is entered. The second word points to the *cost centre* responsible for building the closure. Following this is a block of words containing pointers and a block containing non-pointers. The distinction between the two is for garbage collection purposes.

Heap-allocated closures are classified in two different ways:

1. Which part of the program created the closure, as indicated by the attached cost centre.
2. What kind of closure it is, as described by the additional information stored in the info-table.

We have already discussed cost centres in detail, we now consider the different closure kinds. We first classify the closures into distinct *classes*. Within this we provide a more detailed description string, based on source code identifiers. We have:

Constructors — Evaluated data closures. They are described by the actual data constructor used to build the data object in the source language. This can always be determined at compile time. An alternative description is the data type constructor. This would group more information together.

Manifest functions — Lambda-forms with a non-empty argument list. They are described by the name given to the declaration. If this name not known, as in the case of an anonymous definition, it is given a description "UNKNOWN". For example

```
f x y = ...
```

is given the description "`f`".

Partial applications — Partially applied manifest function. Partial applications are built at runtime. There is only one info-table for partial applications having the description "PAP".

Thunks — Lambda-forms with an empty argument list, i.e. unevaluated closures. These are described by the name of the function being applied in the suspended computation. Again if this name is not known an "UNKNOWN" description string is used. For example

```
t = f a b
```

is also given the description "`f`", but this is an application of `f` not the declaration.

Thus, for any heap object we can identify the cost centre responsible for building it and the kind of closure it represents.

5.2 Aggregate Information

Aggregate profiles provide aggregate information about the behaviour of different pieces of the source program over the entire run of the program. For each cost centre we collect:

Time:

At regular intervals, of approximately 20ms, the clock generates a signal which is fielded by a signal handler. The signal handler simply increments the time counter of the current cost centre. Dividing this count by the total number of interrupts gives the proportion of time spent evaluating instances of the expression.

During garbage collection and other activities of the run time system this timer interrupt is turned off.

Memory allocation:

Whenever we allocate heap memory we increment the memory counter of the current cost centre. This indicates the amount of memory allocated during the evaluation of the expression instances.

It is possible to provide a more detailed breakdown of this information. For each cost centre we can record the allocation required for building different types of closure. An example breakdown might by constructor, function, partial application, and thunk.

Number of instances:

The number of instances of the expression that were evaluated. The scc construct increments the counter of the cost centre it is setting. This count can be used to average the aggregate costs.

Note that this is not the number of times a particular cost centre becomes the current cost centre. This may occur many times for each expression instance — every time a bit of the result is demanded by entering an unevaluated closure created by this cost centre.

5.3 Serial Information

Serial profiles show the way in which the programs resource usage evolves over time. Information is provided for each time interval. This serial information can be of two forms:

1. Aggregate information, such as that described in section 5.2, collected for each time interval.
2. A *sample* of the state of execution taken during the interval.

At the end of each interval execution is suspended. Information aggregated over the interval is recorded and any state samples taken.

We are particularly interested in examining the heap requirements of the programs over time. A heap sample produces a summary of which cost centres and/or closure kinds are responsible for the closures currently live in the heap. When execution is complete these summary profiles can be processed to produce various graphical visualisations of the heap's behaviour over time (Runciman & Wakeling [1992]).

6 Related Work

This work is closely related to work done by Runciman and Wakeling. In Runciman & Wakeling [1991] they identify many of the problems profiling lazy functional languages and proposed an interpretive profiler based on source level graph reduction.

More recently they have proposed and implemented a heap profiling scheme that is similar to ours (Runciman & Wakeling [1992]). They store the *function* that produced the graph node and the name of the *construction* in each closure and produce time profiles of the heap broken up by function and/or construction. Interpreting the profiles of a particular program revealed interesting phenomena about its execution. These lead to improvements being made to the program under investigation and the evaluation scheme used by the compiler.

Our current cost centres are similar to a technique proposed by Appel, Duba and MacQueen for SML (Appel, Duba & MacQueen [1988]). They keep track of the current function being evaluated by adding appropriate statements to the intermediate code. The meter of the current function is incremented at regular timer interrupts providing an aggregate time profile of the different functions. Their work was motivated by the need to keep track of interleaved execution arising from transforming optimisations. Though this is also an issue for us: we arrived at our scheme because we can't avoid the interleaving arising from lazy evaluation.

Putting these two schemes together results in something with very similar capabilities to ours. The main contribution that we make is to move away from the function as the unit of source code. By identifying expressions we provide a considerably more flexible scheme. In particular we can simulate a function scheme by getting the compiler to annotate each function body with a cost centre of the same name. Alternatively we can distinguish between different instances of the same function by annotating the call sites with different cost centres. More importantly it allows us to aggregate a big nest of function calls together. This enables the overall performance of large parts of a program to be compared without being swamped by detailed function oriented information.

7 Future Work

This work is still in an early stage of development. We are still in the process of implementing cost centres and the associated statistics gathering. Once this is done we will set about evaluating the profiling tool.

We are particularly interested in evaluating the cost centre model. Though we do believe that a tool which identifies source by function is adequate for small programs, we do not believe that such a tool is appropriate for large programs. The number of functions, call sites and amount of information is too cumbersome to interpret. We are planning to profile the Glasgow Haskell compiler, a large Haskell program, compiling parts of itself, as a practical test of the profiling tool. We hope this will allow us to:

- Tune the currently poor performance of the compiler.
- Identify any problems with the evaluation techniques used by the underlying implementation of the compiler.
- Evaluate the use of cost centres vs. functions to identify source.
- Identify techniques which assisted the profiling task.
- Identify any enhancements or additional information which would have aided the profiling task if been available to us.

Bibliography

AW Appel, BF Duba & DB MacQueen [Nov 1988], "Profiling in the presence of optimization and garbage collection," SML Distribution.

SL Graham, PB Kessler & MK McKusick [1983], "An execution profiler for modular programs," *Software — Practice and Experience* 13, 671–685.

John Hughes [Apr 1989], "Why functional programming matters," *The Computer Journal* 32.

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

SL Peyton Jones [1992], "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," Dept of Computer Science, University of Glasgow.

C Runciman & D Wakeling [1991], "Proposals and problems for time and space profiling of functional programs," in *Proceedings of the 1990 Glasgow Workshop on Functional Programming, Ullapool*, SL Peyton Jones, G Hutton & CK Holst, eds., Workshops in Computing, Springer Verlag.

C Runciman & D Wakeling [April 1992], "Heap profiling of lazy functional languages," Technical Report 172, Dept of Computer Science, University of York.

B Zorn & P Halfinger [1988], "A memory allocation profiler for C and LISP programs," in *USENIX 88, San Francisco*, 223–237.

A Operational Semantics

We now formally define the manipulation of cost centres in terms of the operational semantics of the STG-machine as presented in Peyton Jones [1992]. This is given for the *STG language*, a rather austere functional langauge.

The following extensions are made to the state transition system:

- The current cost centre, *cc*, is added as an extra element to the machine state.
- All heap closures have a cost centre attached to them. This is indicated by prefixing the heap object with the cost centre.
- Update frames are augmented with the cost centre to be restored once the closure evaluated and the update has been performed.
- A second form of update frame is introduced which is used to keep track of cost centres when no update is actually required. It does not contain a closure to update, just a cost centre to restore.

The STG language is also modified slightly, with the non-update closures (`\n`) being split into two distinct cases: those which are already HNF (`\r`) and those which are not (`\s`). So we have the following update flags:

- `\u` — *Updatable.*
  Unevaluated closures which will be updated with their normal form.

  The cost of evaluating such a closure must be attributed to the cost centre which was responsible for its creation. This is the cost centre which was active when the particular computation was suspended in a thunk ready for later evaluation if demanded. It is stored with the closure.

  We store the current cost centre of the demanding computation in the update frame so it can be restored when the update occurs, indicating that the evaluation of this closure is complete, and the demanding computation resumes.

- `\s` — *Single-entry.*
  Unevaluated closures which promise that they will only be entered at once. They are not updated with their normal form.

  Even though the closure is not updated there is an arbitrary amount of work involved in evaluating it. We must attribute this to the appropriate cost centre. A cost centre update frame is used to store and restore the cost centre of the demanding computation. This does not actually perform an update.

- `\r` — *Reentrant.*
  Closures which are already evaluated and may be entered (and re-evaluated) more than once. Manifest functions, constructors, and partial applications are always reentrant. They are never updated.

  The (small) cost of entering these closures to extract the value within will be attributed to the entering closure, as it is the one who is demanding the value.

A.1 State Transition Rules

We now give the modified state transition rules. Most of the transition rules remain the same, except for the addition of the current cost centre to the state. We concentrate on the rules which manipulate cost centres. All the rules which do not manipulate cost centres are omitted for brevity. Where appropriate the numbers used correspond directly with those used in Peyton Jones [1992].

Evaluating an scc expression

We first give the state transition rule for the evaluation of the new scc expression:

$$\boxed{\begin{array}{llllllll} & \mathit{Eval}\ (\texttt{scc}\ cc_{scc}\ e)\ \rho & as & rs & us & h & \sigma & cc \\ \Longrightarrow & \mathit{Eval}\ e\ \rho & \{\} & \{\} & (as, rs, cc) : us & h & \sigma & cc_{scc} \end{array}}$$

This sets the current cost centre to that indicated by the the scc expression, cc_{scc}. A cost centre update frame, containing the old cost centre, cc, is pushed onto the update stack. It will be restored once the evaluation of e is complete.

Constructing heap objects

Heap objects are constructed by let(rec) expressions. They must have the current cost centre attached to them.

$$(3)\ \boxed{\begin{array}{llllllllll} & \mathit{Eval} & \left(\begin{array}{lll} \texttt{let} & x_1 \texttt{ = } vs_1\ \texttt{\\}u_1\ xs_1 \texttt{ -> } e_1 \\ & \ldots \\ & x_n \texttt{ = } vs_n\ \texttt{\\}u_n\ xs_n \texttt{ -> } e_n \\ \texttt{in}\ e \end{array}\right) & \rho & as & rs & us & h & \sigma & cc \\ \Longrightarrow & e\ \rho' & & & as & rs & us & h' & \sigma & cc \\ \text{where} & \rho' & = & \rho[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] \\ & h' & = & h\left[\begin{array}{l} a_1 \mapsto cc\ (vs_1\ \texttt{\\}u_1\ xs_1 \texttt{ -> } e_1)\ (\rho_{rhs}\ vs_1) \\ \ldots \\ a_n \mapsto cc\ (vs_n\ \texttt{\\}u_n\ xs_n \texttt{ -> } e_n)\ (\rho_{rhs}\ vs_n) \end{array}\right] \\ & \rho_{rhs} & = & \rho \end{array}}$$

The rule for letrec is almost identical, except that ρ_{rhs} is defined to be ρ' instead of ρ.

Entering closures

Entering an evaluated closure (indicated by the \r update flag) does not modify the current cost centre. There is no work required to evaluate them so no costs to attribute to the closures cost centre. Any cost associated with returning the value contained within is attributed to the entering cost centre.

$$
(2)\quad
\begin{array}{|llllllll|}
\hline
 & \mathit{Enter}\ a & as & rs & us & h[a \mapsto cc_{enter}\ (vs\ \mathtt{\backslash r}\ xs\ \mathtt{->}\ e)\ ws_f] & \sigma & cc \\
\text{such that} & \multicolumn{7}{l|}{\mathit{length}(as) \geq \mathit{length}(xs)} \\
\Longrightarrow & & & & & & & \\
 & \mathit{Eval}\ e\ \rho & as' & rs & us & h & \sigma & cc \\
\text{where} & \multicolumn{7}{l|}{ws_a \mathbin{+\!\!+} as' \quad = \quad as} \\
 & \multicolumn{7}{l|}{\mathit{length}(ws_a) \quad = \quad \mathit{length}(xs)} \\
 & \multicolumn{7}{l|}{\rho \quad = \quad [vs \mapsto ws_f,\ xs \mapsto ws_a]} \\
\hline
\end{array}
$$

When entering unevaluated closures, or thunks, the current cost centre most be modified so that the costs of evaluating this closure are attributed to the appropriate cost centre — that attached to the closure. The old cost centre must be restored once the closure has been reduced to HNF. For updatable closures (`\u` update flag) the old cost centre is added to the update frame.

(15)

$$
\begin{array}{|lllllll|}
\hline
 & \mathit{Enter}\ a & as & rs & us & h[a \mapsto cc_{enter}\ (vs\ \mathtt{\backslash u}\ \mathtt{\{\}}\ \mathtt{->}\ e)\ ws_f] & \sigma\quad cc \\
\Longrightarrow & \mathit{Eval}\ e\ \rho & \{\} & \{\} & (as, rs, cc, a) : us & h & \sigma\quad cc_{enter} \\
\text{where} & \multicolumn{6}{l|}{\rho = \rho_{init}[vs \mapsto ws_f]} \\
\hline
\end{array}
$$

A new rule is added for single-entry closures (`\s` update flag). Though this closure does not require updating, we still have to restore the current cost centre when evaluation is complete so a cost centre update frame is pushed.

(15')

$$
\begin{array}{|lllllll|}
\hline
 & \mathit{Enter}\ a & as & rs & us & h[a \mapsto cc_{enter}\ (vs\ \mathtt{\backslash s}\ \mathtt{\{\}}\ \mathtt{->}\ e)\ ws_f] & \sigma\quad cc \\
\Longrightarrow & \mathit{Eval}\ e\ \rho & \{\} & \{\} & (as, rs, cc) : us & h & \sigma\quad cc_{enter} \\
\text{where} & \multicolumn{6}{l|}{\rho = \rho_{init}[vs \mapsto ws_f]} \\
\hline
\end{array}
$$

Updating closures and cost centres

Finally we have to give the rules for updating. These fall into two categories.

- Those that resulted from entering an updatable closure and require the closure to be updated and the cost centre to be restored.
- Those that resulted from entering a single-entry closure or evaluating an `scc` expression and simply require the cost centre to be restored.

We start with the full updates. These must update the closure with the result of evaluating the expression. What cost centre should be given to this new closure? We can choose from the demanding cost centre (stored in the update frame) or the evaluating cost centre (the current cost centre). There may be many possible demanding cost centres. The one which actually demanded it first and forced evaluation is dependent on the implicit evaluation order. To avoid any confusion we attach the evaluating cost centre to the updated closure as it is unique. This decision does not affect further evaluation as the updated closure is in HNF. What it does affect is the cost centre this heap closure will be attributed to.

Now for the update transition rules. There are two possibilities. A constructor that sees an empty return stack:

$$(16)\quad \boxed{\begin{array}{llllllll}
 & \mathit{ReturnCon}\ c\ ws & \{\} & \{\} & (as_u, rs_u, cc_u, a_u) : us & h & \sigma & cc \\
\Longrightarrow & \mathit{ReturnCon}\ c\ ws & as_u & rs_u & us & h_u & \sigma & cc_u \\
\text{where} & vs \text{ is a sequence of arbitrary distinct variables} & & & & & & \\
 & \mathit{length}(vs) = \mathit{length}(ws) & & & & & & \\
 & h_u = h[a_u \mapsto cc\ (vs\ \texttt{\textbackslash r}\ \{\}\ \texttt{->}\ c\ vs)\ ws] & & & & & &
\end{array}}$$

A lambda abstraction which does not have enough arguments on the stack:

$$(17a)\quad \boxed{\begin{array}{lllllll}
 & \mathit{Enter}\ a & as & \{\} & (as_u, rs_u, cc_u, a_u) : us & h & \sigma\ \ cc \\
\text{such that} & h\ a = cc_{enter}(vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & & & & & \\
 & \mathit{length}(as) < \mathit{length}(xs) & & & & & \\
\Longrightarrow & & & & & & \\
 & \mathit{Enter}\ a & as \mathbin{+\!\!+} as_u & rs_u & us & h_u & \sigma\ \ cc_u \\
\text{where} & xs_1 \mathbin{+\!\!+} xs_2 = xs & & & & & \\
 & \mathit{length}(xs_1) = \mathit{length}(as) & & & & & \\
 & f \text{ is an arbitrary variable} & & & & & \\
 & h_u = h[a_u \mapsto cc\ ((f : xs_1)\ \texttt{\textbackslash r}\ \{\}\ \texttt{->}\ f\ xs_1)\ (a : as)] & & & & &
\end{array}}$$

We have a corresponding pair of rules for cost centre updates. Here, the update frame only contains the cost centre to be restored, cc_u. There is no closure to be updated so the heap is left unchanged. For constructors we have:

$$(16')\quad \boxed{\begin{array}{llllllll}
 & \mathit{ReturnCon}\ c\ ws & \{\} & \{\} & (as_u, rs_u, cc_u) : us & h & \sigma & cc \\
\Longrightarrow & \mathit{ReturnCon}\ c\ ws & as_u & rs_u & us & h & \sigma & cc_u
\end{array}}$$

And lambda abstraction without enough arguments:

$$(17a')\quad \boxed{\begin{array}{lllllll}
 & \mathit{Enter}\ a & as & \{\} & (as_u, rs_u, cc_u) : us & h & \sigma\ \ cc \\
\text{such that} & h\ a = cc_{enter}(vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & & & & & \\
 & \mathit{length}(as) < \mathit{length}(xs) & & & & & \\
\Longrightarrow & & & & & & \\
 & \mathit{Enter}\ a & as \mathbin{+\!\!+} as_u & rs_u & us & h & \sigma\ \ cc_u
\end{array}}$$

Hidden execution

Certain parts of the implementation are not revealed by this operational semantics. In particular the use of indirections when updating is hidden. The semantics simply define a new heap mapping. Indirections themselves are ignored when profiling as they are removed during garbage collection.

When processing a stack of contiguous update frames we indirect many updates to a single closure. Each update frame may have a different cost centre but we can only give the resulting closure a single cost centre. We give this closure the cost centre which was active just before the first update as this was the cost centre which actually evaluated the expression. This turns out to be a little tricky to do as the closure may be returned in registers and not actually built until after all the update frames have been removed and the cost centres restored. This is overcome by requiring the register return convention to also return the closures cost centre in a register. The updated closure can now be built with the required cost centre — that returned in the register. Any allocation should also be attributed to this cost centre.