Time and space profiling for non-strict, higher-order functional languages

Patrick M. Sansom
Dept. of Computing Science,
University of Glasgow,
Glasgow, Scotland
sansom@dcs.glasgow.ac.uk

Simon L. Peyton Jones
Dept. of Computing Science,
University of Glasgow,
Glasgow, Scotland
simonpj@dcs.glasgow.ac.uk

Abstract

We present the first profiler for a compiled, non-strict, higher-order, purely functional language capable of measuring *time* as well as *space* usage. Our profiler is implemented in a production-quality optimising compiler for Haskell, has low overheads, and can successfully profile large applications.

A unique feature of our approach is that we give a formal specification of the attribution of execution costs to cost centres. This specification enables us to discuss our design decisions in a precise framework. Since it is not obvious how to map this specification onto a particular implementation, we also present an implementation-oriented operational semantics, and prove it equivalent to the specification.

1 Motivation and overview

Everyone knows the importance of profiling tools: the best way to improve a program's performance is to concentrate on the parts of the program which are eating the lion's share of the total space and time resources. One would expect profiling tools to be particularly useful for very high-level languages — such as non-strict, higher-order languages — where the mapping from source code to target machine is much less obvious to the programmer than it is for (say) C. Despite this obvious need, profiling tools for such languages are very rare. Why? Because profilers can only readily measure or count low-level execution events, whose relationship to the original high-level program is far from obvious.

With this in mind, we have developed a profiler for Haskell, a higher-order non-strict, purely functional language (Hudak et al. [1992]). We make three main contributions, all of which relate to the question of mapping low-level execution events onto the programmers-eye view:

1. *We describe the first profiler for a compiled, non-strict language capable of measuring execution time as well as space usage.* Non-strict languages are usually implemented using some form of *lazy evaluation*, whereby values are computed only when required. For example, if one function produces a list which is consumed by another, execution will alternate between producer and consumer in a co-routine-like fashion. Execution of different parts of the program is therefore finely interleaved, which makes it difficult accurately to measure how much time is spent in each part of the program. Our profiler solves this problem (Section 2.2); indeed, its results are entirely independent of evaluation order.

2. *We support the profiling of large programs by allowing the systematic subsumption of execution costs under programmer-controlled headings, or "cost centres".* Most profilers implicitly attribute costs to the function or procedure which incurred them. This is unhelpful for languages like Haskell, that encourage a modular programming style based on a large number of small, often heavily re-used, functions, for two reasons: firstly, there are simply too many functions in a large program; and secondly it does not help much to be told (say) that the program spends 20% of its time in the heavily-used library function `map`. Separating the notion of a cost centre from that of a procedure allows us to avoid these difficulties (Section 2.1).

3. *We provide a formal specification of the attribution of execution costs to cost centres.* Higher-order languages make it harder to give the programmer a clear model of where costs are attributed. For example, suppose a function produces a data structure with functions embedded in it. Should the execution costs of one of these embedded functions be attributed to the "place" where it is called, or to the function which produced the data structure?

   A unique contribution of this paper is that we back up our informal description of cost attribution (Section 2.2) with a formal specification, which we call a *cost semantics* (Section 2.3). In the framework thus created we are able to explore the design space in a precise way (Section 4).

While our approach can handle non-strict languages such as Haskell, it is not restricted to them: it can also accommodate strict languages such as SML (though many of the issues we address here would be simplified).

To appear in 22nd ACM Symposium on Principles of Programming Languages, San Francisco, Jan 1995.
The full version of this paper, which includes the correctness proof, is available as Research Report FP-1994-10, Dept of Computing Science, University of Glasgow, Nov 1994.

From a practical point of view, our technique is easy to implement. In Section 3 we describe a full implementation of the profiler in the Glasgow Haskell Compiler (Peyton Jones et al. [1993]), a state-of-the-art optimising compiler for Haskell. As well as an informal description of the implementation, we present a state-transition system which describes it formally, and prove it equivalent to the specification

The run-time overheads of the profiler are low (less than a factor of 2), even compared with fully-optimised compiled code, so that the profiler can be used to instrument even very large programs (Section 5).

2 Specifying the profiler

For a profiler to be useful it must be possible to explain to a programmer the exact way in which execution costs are attributed to the headings under which the profiler reports them — we must give a specification of the profiler. For a higher-order, non-strict language such as Haskell, we have found that this specification is remarkably slippery. Every time we came up with an informal specification we found new examples for which the specification was inadequate!

This experience eventually led us to develop a formal specification of the way in which our profiler attributes costs. In this section we give an informal model of cost attribution, show how it is inadequate, and then describe our formal model.

An important constraint is that the profiler should do "what the programmer expects" in commonly occurring cases. The formal specification is useful for obscure or difficult cases, but our goal is that most of the time programmers should not need to refer to it. In this sense, the formal system plays exactly the same role as the formal semantics of a programming language: it is a guide to implementors, and the final arbiter of obscure cases.

2.1 Cost centres

In the discussion above we have repeatedly referred to the "part" of a program to which execution costs should be attributed. Most profilers implicitly identify such "parts" with functions or procedures. As we have already argued, such an identification is problematic for functional languages. Instead, we introduce a separate notion of a *cost centre*, to which execution costs are attributed. For example, consider the following function definitions:

```
my_fun xs = scc "mapper" (map square xs)
square x  = x * x
```

The `scc` ("set-cost-centre"[1]) construct explicitly annotates an expression with a cost centre to which the costs of evaluating that expression should be attributed. In this case, the costs of evaluating `(map square xs)` will be attributed to the (arbitrarily-named) cost centre `"mapper"`.

[1]The irony of this imperative-sounding name is not lost on us.

`scc` is a language construct, like `let` or `case`, and not a function. The cost centre is a literal string, not a computed value, the scope of the `scc` extends as far to the right as possible. The `scc` annotations can be added either manually by the programmer, or automatically by the compiler (see Section 3.2).

2.2 Cost attribution

The crux of the matter is, of course, exactly which costs are attributed to which cost centre. In general, given an expression `scc` cc exp, the costs attributed to cc are the entire costs of evaluating the expression exp as far as the enclosing context demands it, excluding

(a) the cost of evaluating the free variables of exp, and

(b) the cost of evaluating any `scc`-expressions within exp (or within any function called from exp).

This definition has the following consequences:

- **Costs are aggregated.** In the example above, the function `square` does not have an `scc` construct, so its costs are attributed to the cost centre of its caller, in this case `"mapper"`. (Other calls to `square` will be attributed to the cost centre of their callers, not to `"mapper"` of course.) Similarly, the cost of executing this call to the library function `map` will be attributed to `"mapper"` as well. In short, except where explicit `scc`s specify otherwise, the costs of callees are automatically subsumed into the costs of the caller.

- **Results are independent of evaluation order.** When `my_fun` is called, its argument `xs` may not be fully evaluated, and its further evaluation may ultimately be forced by `map` called from within `my_fun`. Nevertheless, the costs of evaluating `xs` will not be attributed to `"mapper"` but rather to the cost centre which encloses the producer of `xs`. Similarly, the result of `my_fun` is a list, which may be evaluated fully, partially, or not at all. The costs of whatever evaluation is performed will be attributed to `"mapper"`, no more and no less.

  In effect, this means that the programmer does not need to understand the program's evaluation order in order to reason about which costs are attributed to which cost centre.

- **The degree of evaluation performed is unaffected.** The result of `my_fun` will be evaluated no more and no less than would be the case in an un-profiled program.

  It follows that the costs attributed to `"mapper"` will depend on how much of `my_fun`'s result is evaluated by its caller. To say that the results are independent of evaluation *order* (as we just noted above) is not to say that they are independent of evaluation *degree*!

- **Costs are attributed to precisely one cost centre.** Using the same statistical inheritance techniques as **gprof** (Graham, Kessler & McKusick [1983]), it is

$$
\begin{array}{rcl}
e & ::= & \lambda x.e \\
 & | & e\ x \\
 & | & x \\
 & | & \texttt{let}\ x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n\ \texttt{in}\ e \\
 & | & C\ x_1 \cdots x_a \\
 & | & \texttt{case}\ e\ \texttt{of}\ \{C_j\ x_{j1} \cdots x_{ja_j}\ \texttt{->}\ e_j\}_{j=1}^{n} \\
 & | & \texttt{scc}\ cc\ e \\
prog & ::= & x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n
\end{array}
$$

Figure 1: Language Syntax

> possible to attribute costs both to the "current" cost centre and any "enclosing" cost centres. However, this requires a record of the current cost centre *and its enclosing cost centre* to be maintained and attached to every heap closure (see Sansom [1994b]). We have not implemented this due to the added complexity and overhead. Instead we rely on the simple, but accurate, mechanism for aggregating un-profiled costs described above.

Despite our attempt at precision, our definition is still vague, especially when higher-order functions are concerned. For example, what costs are attributed to `"tricky"` in the expression `scc "tricky" (\x -> square x)`? Zero cost, because the λ-abstraction is already in head normal form? Or are the costs of every application of the λ-abstraction attributed to `"tricky"`? Would the answer be different if the expression were `scc "tricky" (f y)`, where `(f y)` evaluates, perhaps after many reductions, to a function? What if it were `scc "tricky" (f y, 2)`, where the function is embedded in a pair? In order to answer questions like these precisely we developed a formal model for cost attribution, which we discuss next.

2.3 Cost semantics

In order to speak precisely about cost attribution we need a more formal model for the costs themselves. We base our model on Launchbury's operational semantics for lazy graph reduction (Launchbury [1993a]; Sestoft [1994]), augmenting it with a notion of cost attribution.

2.3.1 Language

The language we treat is given in Figure 1[2]. It looks at first like no more than a slightly-sugared lambda calculus, with mutually-recursive `let`s, constructors, `case`, and `scc`, but it has an important distinguishing feature: *the argument of a function application is always a simple variable.* A non-atomic argument is handled by first binding it to a variable using a `let` expression. This has a direct operational interpretation: a non-atomic function argument must be constructed in the heap (the `let`-expression) before the function is called (the application).

[2]The language is a close cousin of the STG language described in Peyton Jones [1992], and is a subset of the Core language used in the Glasgow Haskell compiler.

For notational convenience we use the abbreviation $\{x_i = e_i\}$ for the set of bindings $x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n$. Similarly we write $[y_i \mapsto e_i]$ for the finite mapping $[y_1 \mapsto e_1, \ldots, y_n \mapsto e_n]$ and $e[e_i/x_i]$ for the substitution $e[e_1/x_1, \ldots, e_n/x_n]$. We also abbreviate $x_1 \cdots x_a$ with $\overline{x}_a$ and drop the ${}_{j=1}^{n}$ in the `case` alternatives. We use $\equiv$ for syntactical identity of expressions.

2.3.2 The judgement form

We express judgements about the cost of evaluating an expression thus:

$$cc, \Gamma : e \ \Downarrow_\theta \ \Delta : z, cc_z$$

This should be read: "In the context of the heap Γ and cost-centre cc, the expression e evaluates to the value z, producing a new heap Δ and cost centre cc_z; the costs of this evaluation are described by θ." To make sense of this, we need the following vocabulary:

- A *value*, z, is either a λ-abstraction or a (saturated) constructor application.
- A *cost centre*, cc, is a name to which costs are attributed.
- A *heap*, Γ, Δ, Θ, is a finite mapping from variable names to pairs of an expression and a cost centre. The notation

  $$\Gamma[x \overset{cc}{\mapsto} e]$$

  means the heap Γ extended by a mapping from x to the expression e annotated with cost centre cc.

  In general, the cost centre attached to a binding is the cost centre which enclosed that binding. It is used for two purposes: to ensure correct cost attribution when a thunk is evaluated, and to give cost-centre attribution when a heap census is taken.

- A cost-attribution, θ, is a finite mapping of cost centres to costs. It simply records the costs attributed to each cost centre. Cost attributions are combined using $\uplus$ which determines the total cost attributed to each cost centre. The costs themselves are denominated in arbitrary units, as follows:

Cost of	Denoted
Application	A
Case expression	C
Evaluating a thunk	V
Updating a thunk	U
Allocating a heap object	H

  One should not think of these costs as constants. The cost semantics specifies *which* cost centre is attributed with the cost of (say) doing a heap allocation. The semantics makes no attempt to specify exactly *how much* cost is so attributed; it just says "H". The actual implementation attributes the *actual* execution costs (of time and space) to the cost centre specified by the semantics; it does not count A's, C's, and so on, at all.

$$cc, \Gamma : \lambda y.e \;\Downarrow_{\{\}}\; \Gamma : \lambda y.e, cc \qquad \textit{Lambda}$$

$$cc, \Gamma : C\ \overline{x}_a \;\Downarrow_{\{\}}\; \Gamma : C\ \overline{x}_a, cc \qquad \textit{Constructor}$$

$$\frac{cc, \Gamma : e \;\Downarrow_{\theta_1}\; \Delta : \lambda y.e', cc_\lambda \qquad \boxed{\begin{matrix} cc \\ cc_\lambda \end{matrix}}, \Delta : e'[x/y] \;\Downarrow_{\theta_2}\; \Theta : z, cc_z}{cc, \Gamma : e\ x \;\Downarrow_{\{cc \mapsto \mathrm{A}\} \uplus \theta_1 \uplus \theta_2}\; \Theta : z, cc_z} \qquad \textit{Application}$$

$$cc, \Gamma[x \overset{cc_z}{\mapsto} z] : x \;\Downarrow_{\{cc \mapsto \mathrm{V}\}}\; \Gamma[x \overset{cc_z}{\mapsto} z] : z, \mathrm{SUB}(cc_z, cc) \qquad \textit{Var(whnf)}$$

$$\textbf{where} \quad \begin{aligned} \mathrm{SUB}(\texttt{"SUB"}, cc) &= cc \\ \mathrm{SUB}(cc_z \quad, cc) &= cc_z \end{aligned}$$

$$\frac{cc_e, \Gamma : e \;\Downarrow_\theta\; \Delta : z, cc_z}{cc, \Gamma[x \overset{cc_e}{\mapsto} e] : x \;\Downarrow_{\{cc \mapsto \mathrm{V}\} \uplus \theta \uplus \{cc_z \mapsto \mathrm{U}\}}\; \Delta[x \overset{cc_z}{\mapsto} z] : z, cc_z}\; e \not\equiv z \qquad \textit{Var(thunk)}$$

$$\frac{cc, \Gamma[y_i \overset{cc}{\mapsto} e_i[y_i/x_i]] : e[y_i/x_i] \;\Downarrow_\theta\; \Delta : z, cc_z}{cc, \Gamma : \texttt{let}\ \{x_i \texttt{ = } e_i\}\ \texttt{in}\ e \;\Downarrow_{\{cc \mapsto n*\mathrm{H}\} \uplus \theta}\; \Delta : z, cc_z}\; y_i\ \textbf{fresh} \qquad \textit{Let}$$

$$\frac{cc, \Gamma : e \;\Downarrow_{\theta_1}\; \Delta : C_k\ \overline{x}_{a_k}, cc_C \qquad cc, \Delta : e_k[x_i/y_{ki}] \;\Downarrow_{\theta_2}\; \Theta : z, cc_z}{cc, \Gamma : \texttt{case}\ e\ \texttt{of}\ \{C_j\ \overline{y_j}_{a_j}\ \texttt{->}\ e_j\} \;\Downarrow_{\{cc \mapsto \mathrm{C}\} \uplus \theta_1 \uplus \theta_2}\; \Theta : z, cc_z} \qquad \textit{Case}$$

$$\frac{cc_{scc}, \Gamma : e \;\Downarrow_\theta\; \Delta : z, cc_z}{cc, \Gamma : \texttt{scc}\ cc_{scc}\ e \;\Downarrow_\theta\; \Delta : z, cc_z} \qquad \textit{SCC}$$

Figure 2: Formal Cost Semantics

The cost, θ_{MAIN}, of evaluating the whole program is obtained from the judgement

$$\texttt{"MAIN"}, \Gamma_{init} : \texttt{main} \;\Downarrow_{\theta_{MAIN}}\; \Delta : z, cc_z$$

The initial cost centre is "MAIN", to which all costs are attributed except where an `scc` construct specifies otherwise. The initial heap, Γ_{init}, binds each top-level identifier to its right-hand side. What cost-centre should be associated with these bindings? A top-level binding defines either a *function* (if it has arguments) or a *constant applicative form* (if it does not). The costs of top-level functions should be subsumed by their caller, so we give their bindings in Γ_{init} the special pseudo-cost-centre "SUB" to indicate this fact. The "SUB" cost centre is treated specially by the rules which follow. We discuss the treatment of constant applicative forms later, in Section 4.1.

2.3.3 The rules

The cost-augmented semantics is given in Figure 2. The following paragraphs discuss the rules.

The cost centre on the left hand side of the judgement is the "current cost centre", to which the costs of evaluating the expression should be attributed. The last rule, *SCC*, is easy to understand: it simply makes the specified cost centre into the current cost centre.

It is less obvious why we need a cost centre on the right hand side, which we call the "returned cost centre". The *Lambda* and *Constructor* rules show where it comes from: in both cases the expression to be evaluated is in head normal form, so it is returned, along with the current cost centre. (The other rules simply propagate it.) What use is made of the returned cost centre? To see this we must look at the two rules which "consume" head normal forms, namely *Application* (where a function is evaluated before applying it), and *Case* (where a data value is evaluated before taking it apart):

- In the *Case* rule the returned cost centre cc_C is simply ignored. The appropriate alternative is chosen, and its right-hand side is evaluated in the original cost centre enclosing the `case`. That is as one would expect: the costs of evaluating the alternatives accrue to the cost centre enclosing the `case` expression.
- In the *Application* rule, though, there is an interesting choice to be made. The function, e, is evaluated, delivering (presumably) a λ-abstraction $\lambda y.e'$. The question now arises: should the body of the abstraction be evaluated in the cost centre returned by evaluat-

ing the abstraction cc_λ, or in the cost centre enclosing the application, cc? This choice is represented by the box in the *Application* rule, and is discussed further in Section 2.4.

Lastly, we deal with the *Let* and *Variable* rules, which concern the construction and evaluation of heap-allocated thunks. The *Let* rule extends the heap with bindings for newly-allocated closures. The y_i are freshly-chosen names, directly modelling heap addresses, and are substituted for the corresponding x_i throughout. This substitution ensures that two instantiations of the same `let`-expression don't interfere with each other by binding the same variable twice.

The current cost centre is pinned on each binding created by the *Let* rule. The two *Variable* rules shows how this cost centre is used:

- When a variable is to be evaluated, and it is bound to a value, the *Var(whnf)* rule says that the value is returned, with an unchanged heap, and the returned cost-centre is that pinned on the binding, cc_z. Unless, that is, cc_z is `"SUB"`, in which case the current cost centre is returned, which achieves the effect of subsuming the costs of top-level functions into their callers.
- The *Var(thunk)* rule deals with the situation when the variable is not bound to a value. In this case, the expression to which the variable is bound, is evaluated in the context of the cost centre pinned on the binding, cc_e. The newly calculated value is recorded in the resulting heap, and the costs of entering the thunk and updating it (V and U) are recorded in the cost attribution. Notice that the new binding, of the variable to its value, has the returned cost centre cc_z pinned on it, and also that the cost of the update is attributed to cc_z. This is the second way in which the returned cost centre is used.

The crucial point is that these rules give us *a language in which to discuss cost attribution in a precise manner.* For example, should the cost of an update be attributed to cc or cc_z in the *Var(thunk)* rule? One can argue the toss, but at least the rules give us a precise way to state the question and the answer.

2.4 Lexical and evaluation scoping

In this section we expose and discuss the major design decisions we encountered while developing our cost semantics. Consider the expression

```
let f = scc "fun" (\x y -> x*y+1)
in f 23 16
```

To which cost centre should the cost of computing `(23*16+1)` be attributed? There are two alternatives:

Lexical scoping. Attribute the cost to `"fun"`, the cost centre which lexically encloses the `\xy`-abstraction.

Evaluation scoping. Attribute the cost to the cost centre active when the function `f` is applied.

In effect, lexical scoping attributes the cost of executing a function body to the cost centre which encloses the *declaration site*, while evaluation scoping attributes the cost of executing the function body to the cost centre of the function's *application site*. The difference between lexical and evaluation scoping has a precise and elegant manifestation in the semantics of Figure 2: for lexical semantics we choose the cc_λ alternative from the box in the *Application* rule, and for evaluation scoping we choose the cc alternative.

The practical implications of this distinction are not immediately obvious. Consider the expression

```
let f = scc "fun" exp
in (scc "app1" f a) + (scc "app2" f b)
```

Lexical scoping attributes the total cost associated with the declaration of `f` to the cost centre `"fun"`. This includes both the cost of evaluating `f` itself to a λ-abstraction, and the cost of applying `f`. (The cost centres `"app1"` and `"app2"` are only attributed with the small cost of invoking the application.) In contrast, evaluation scoping provides a finer breakdown of costs, attributing the cost of evaluating the function `f` to `"fun"`, and the costs of applying `f` to the respective application sites `"app1"` and `"app2"`.

We have implemented both profiling schemes, and tried them out in practice. Our experience is that lexical scoping usually measures "what the programmer expected", while evaluation scoping often does not. The main reason for this is that evaluation scoping is very sensitive to whether an `scc` construct is placed around the complete application of a function or not; for example, the costs attributed to `"f"` in `(scc "f" f) x` are very different to those attributed to `"f"` in `(scc "f" f x)`. Although this allows evaluation scoping to express distinctions which are inaccessible in lexical scoping, we have found that in practice such distinctions are more of a hindrance than a help.

Section 4 discusses particular situations in which even lexical scoping has unexpected behaviour.

3 Implementation

Perhaps surprisingly, our lexical cost-attribution model can be implemented quite cheaply. We begin our description of the implementation with an informal overview of the measurement technology and the output it produces (Section 3.1). After this we briefly discuss program transformation (Section 3.3), before turning to the question of how we bridge the gap between the (big-step) cost semantics of Section 2.3 and the individual steps taken by the execution mechanism (Section 3.4).

3.1 Implementation overview

Figures 3 and 4 give an example of the output produced by the profiler. The former shows what fraction of the execution time (`%time`) and what fraction of the heap allocation (`%alloc`) was attributed to each cost centre. The latter shows the composition of the (live) heap data, by cost centre, plotted against time, in the style of Runciman & Wakeling

```
Tue May 18 17:03 1993 Time and Allocation Profile
               (Lexical Scoping)

  hsc-0.13  +RTS -H25M -p -RTS  -C -hi ...

total time  =       240.48 secs (12024 ticks@20ms)
total alloc = 619,779,088 bytes
               51,846,277 closures

COST CENTRE      scc subcc  %time %alloc
TypeChecker        1     0   45.4   44.6
Renamer            1     0   25.0   27.0
builtinNames       1     0    7.6   14.4
PrintRealC         1     0    4.5    4.3
Core2Core          1     0    3.9    2.5
MAIN               1     1    2.7    2.7
CodeGen            1     0    1.5    1.1
rdModule           1     0    1.8    1.2
Stg2Stg            1     0    1.1    0.5
FlattenAbsC        1     0    0.7    0.5
cvModule           1     1    0.6    0.5
Core2Stg           1     0    0.6    0.3
  ...
```

Figure 3: Cost Centre Profile of Glasgow Haskell compiler

[1993]. These measurements are collected in the following way:

- At any moment the "current cost centre" is held in a special register. For example, when evaluation of an `scc` expression is begun, the register is loaded with the specified cost centre. A cost centre is represented by a pointer to a block of store which holds all the counters in which the costs attributed to that cost centre are accumulated.
- A regular clock interrupt runs a service routine which increments a tick counter in the current cost centre. This enables the relative cost of the different "parts" of the program to be determined, and reported in the `%time` column of Figure 3. This statistical sampling works in just the same way as any standard Unix profiler; the more clock ticks that have elapsed, the more accurate the results. (The alternative, that of sampling a clock whenever the current cost centre is changed, would have unacceptable overheads, even apart from any worries about the accumulation of systematic quantisation errors.)
- The compiler plants in-line code to increment an allocation counter held in the current cost centre whenever a new heap object is allocated. In this way the profiler is able to record how much heap was allocated by each cost centre; this leads to the `%alloc` column of Figure 3.
- Whenever evaluation of (an instance of) an `scc` expression is begun, the *entry count* of the new cost centre and the *sub-entry count* of the current cost centre are both incremented. These counts, which are displayed in Figure 3 in columns `scc` and `subcc` respectively, provide information similar to the function-call-counts of conventional profilers. The sub-entry count is useful mainly as a reminder that some of the cost of evaluating the expression is attributed to an inner cost centre.

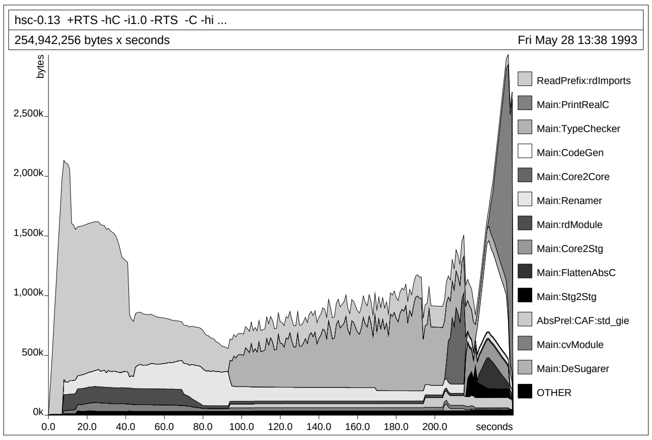

Figure 4: Heap Profile of Glasgow Haskell compiler

- An extra field is added to every heap object. This field is initialised from the current-cost-centre register when the object is allocated. At regular intervals (multiples of a clock tick) a census of all live heap objects is taken. This allows the profiler to produce a *heap profile*, an example of which is given in Figure 4.
- The cost-centre field in heap objects plays an additional role for thunks (suspensions). It records the cost centre which was current when the thunk was allocated. When the thunk is subsequently evaluated, the current-cost-centre register is loaded from the field in the thunk, thus neatly restoring its value at the moment the thunk was created. This is just what one would expect from the *Var(thunk)* rule of Figure 2.

All of this is quite easily done; indeed, one of the beauties of the approach is that the runtime implementation is so simple. What takes a little more care is to make sure that every pass of our optimising compiler respects cost attribution: that is, it must not move costs from one cost centre to another. We turn to this question next.

3.2 Automatic annotation

The use of explicit `scc` annotations allows fine control of the "resolution" of the profiler. At one extreme, the programmer can simply add a handful of `scc` annotations which enclose the main distinct computations in the program. At the other, several individual sub-expressions within a single function definition (already identified as a culprit) can be `scc`'d to determine which is responsible for which costs.

However, many programmers will prefer to profile a completely unmodified program. We support this in a straightforward way by providing a compiler flag which annotates every top-level definition in the module being compiled with an eponymous `scc`. In practice, compiling an entire program with this option is often the first step in finding which part of the program is eating the lion's share of the resources; it can then be followed, if necessary, with a more selective manual placement of `scc` annotations.

3.3 Transformation

Any optimising compiler performs many different program transformations during compilation. For the profiling results to be meaningful it is important that these program transformations maintain the correct attribution of costs. Specifically, *program transformations that move evaluation from the scope of one cost centre to another must be avoided.*

Since cost centre boundaries are explicitly identified by the `scc` construct, the only transformations that have to be curtailed are those that would move costs across an `scc` boundary. Thus the transformations which have to be curtailed depend on the actual `scc` annotations introduced by the programmer. In particular, the optimisation of a large program containing only a few `scc` annotations is largely unaffected by profiling. If instead, every function is annotated with a separate cost centre, inter-procedural optimisation — which is very important in languages which encourage small functions — may have to be curtailed.

However, the `scc` construct also provides us with a language that can be used to express alternative versions of transformations that would otherwise move evaluation from the scope of one cost centre to another. For example, we can move a `let` binding across an `scc` boundary if we annotate the sub-expression which is moved into the scope of a different cost centre with its original cost centre.

```
\y -> scc "y" let x = exp in body
==>
let x = scc_sub "y" exp in \y -> scc "y" body
```

The $\texttt{scc}_{sub}$ annotation is identical to an `scc` annotation, except that it doesn't increment the entry counts. These are only incremented when the original `scc` is evaluated.

The formal cost semantics provides us with a framework in which the correctness of these alternative transformations — with respect to the attribution of costs — can be examined and even proved. For example, in the transformation above we can reason that the transformation does not affect the cost attribution, except by moving the small cost of allocating the closure for `x` to the enclosing cost centre. (In our implementation, we are willing to accept moving small constant amounts of work from one cost centre to another, to allow more transformations to take place than would be the case if we were completely rigorous about not moving costs.)

A full treatment of transformation in the presence of `scc` annotations is beyond the scope of this paper. The interested reader is referred to Sansom [1994b].

3.4 An operational semantics

Having optimised the program, being sure to maintain the appropriate cost attribution, we then have to generate code to execute and profile the program. Unfortunately there is a large gap between the formal cost semantics of Figure 2 and any implementation, because the cost semantics is "big-step", specifying the final result on the right-hand-side of a judgement, and the implementation is necessarily "small-step", specifying intermediate steps in the computation. Sestoft [1994] bridges this gap in his derivation of an abstract machine from Launchbury's natural semantics. The first step in this derivation introduces a stack to record the context (or continuation) of each subtree in the state.

We present a version of Sestoft's stack-based state-transition semantics, augmented with the manipulation of cost centres and a notion of cost attribution for lexical scoping. This enables us to highlight a number of implementation-related design decisions, which are applicable to a number of different abstract machines (see Section 3.6). We also prove that the cost attribution reported by the state-transition semantics is correct with respect to the cost semantics (Section 3.5).

The stack-based state-transition semantics, which includes the manipulation of cost centres for lexical scoping, is given in Figure 5. The state consists of a 5-tuple $(ccc, \Gamma, e, S, \theta)$ where ccc is the current cost centre, Γ is the annotated heap, e is the expression currently being evaluated or the *control*, S is the stack (initially empty), and θ is the cost attribution of the evaluation to date.

The state-transition rules correspond directly to the rules in the cost semantics. There is one state-transition rule which begins the computation for each subtree in the corresponding semantic rule. In addition the *Var(thunk)* rule gives rise to a second state-transition rule which updates the heap with the computed result. The correspondence between the semantic and state-transition rules is summarised below:

Semantic Rule	State Transition Rule(s)
Lambda	n.a.
Application	app_1, app_2
Var(whnf)	var_1
Var(thunk)	var_2, var_3
Let	let
Constructor	n.a.
Case	$case_1$, $case_2$
SCC	scc

The key component of the state-transition rules is the stack. It is used to record any information which is required when evaluation of a subtree is complete. There are three places where this is required:

- In the *Application* rule the argument x is pushed onto the stack while the function expression e is evaluated (rule app_1). When the λ-abstraction is evaluated it retrieves this argument off the stack and evaluates its body (rule app_2).
- In the *Var(thunk)* rule an update marker $\#x$ is pushed onto the stack while the thunk e is evaluated (rule var_2). When evaluation is complete the result value z (a λ-abstraction or constructor) will encounter the update marker on the stack and update the heap (rule var_3).
- In the *Case* rule the alternatives $alts$ is saved on the stack while the case expression e is evaluated (rule $case_1$). When the evaluation of the constructor is complete the appropriate alternative is selected and evaluated (rule $case_2$). We also save the current cost centre on the stack with the alternative since it has to be restored when the alternative is evaluated.

ccc	Heap	Control	Stack	θ	$\Longrightarrow$	ccc	Heap	Control	Stack	Attribution	rule
cc	Γ	$e\ x$	S	θ	$\Longrightarrow$	cc	Γ	e	$x : S$	$\theta \uplus \{cc \mapsto \mathrm{A}\}$	app_1
cc_λ	Γ	$\lambda y.e$	$x : S$	θ	$\Longrightarrow$	cc_λ	Γ	$e[x/y]$	S	θ	app_2
cc	$\Gamma[x \overset{cc_z}{\mapsto} z]$	x	S	θ	$\Longrightarrow$	cc_{sub}	$\Gamma[x \overset{cc_z}{\mapsto} z]$	z	S	$\theta \uplus \{cc \mapsto \mathrm{V}\}$	var_1
cc	$\Gamma[x \overset{cc_e}{\mapsto} e]$	x	S	θ	$\Longrightarrow$	cc_e	Γ	e	$\#x : S$	$\theta \uplus \{cc \mapsto \mathrm{V}\}$	var_2
cc_z	Γ	z	$\#x : S$	θ	$\Longrightarrow$	cc_z	$\Gamma[x \overset{cc_z}{\mapsto} z]$	z	S	$\theta \uplus \{cc_z \mapsto \mathrm{U}\}$	var_3
cc	Γ	$\texttt{let}\ \{x_i \texttt{ = } e_i\}\ \texttt{in}\ e$	S	θ	$\Longrightarrow$	cc	$\Gamma[y_i \overset{cc}{\mapsto} \widehat{e_i}]$	$\widehat{e}$	S	$\theta \uplus \{cc \mapsto n{*}\mathrm{H}\}$	let
cc	Γ	$\texttt{case}\ e\ \texttt{of}\ alts$	S	θ	$\Longrightarrow$	cc	Γ	e	$(alts, cc) : S$	$\theta \uplus \{cc \mapsto \mathrm{C}\}$	$case_1$
cc_C	Γ	$C_k\ \overline{x}_{a_k}$	$(alts, cc) : S$	θ	$\Longrightarrow$	cc	Γ	$e_k[x_i/y_{ki}]$	S	θ	$case_2$
cc	Γ	$\texttt{scc}\ cc_{scc}\ e$	S	θ	$\Longrightarrow$	cc_{scc}	Γ	e	S	θ	scc

In the var_1 rule $cc_{sub} = \mathrm{SUB}(cc_z, cc)$, where SUB is as defined in Figure 2.
In the var_2 rule $e \not\equiv z$.
In the let rule the introduced variables, y_i, must be distinct and fresh. The notation $\widehat{e}$ means $e[y_i/x_i]$.
In the $case$ rules $alts$ stands for the list $\{C_j\ \overline{y_j}_{a_j}\ \texttt{->}\ e_j\}$ of alternatives.
In the $case_2$ rule e_k is the right hand side of the k'th alternative.

Figure 5: Stack-based State Transition Rules for Lexical Scoping

Lest all this seem too obvious, we remark that using evaluation rather than lexical scoping, which involves changing only one rule of the formal cost semantics, requires significant and pervasive changes to the state-transition semantics (the details are in Sansom [1994b]).

3.5 Correctness of the operational semantics

We have proved that the cost attribution reported by the stack-based state-transition semantics for lexical scoping (Figure 5) is correct with respect to the formal cost semantics (Figure 2).

Theorem:
$$(cc, \{\}, e, [], \{\}) \Rightarrow^* (cc', \Delta, z, [], \theta) \quad \text{if and only if} \quad cc, \{\} : e \ \Downarrow_\theta\ \Delta : z, cc'$$

The proof is an extension of the correctness proof of Sestoft's original stack-based state-transition semantics (Sestoft [1994]). Space prohibits us from including it here, but it can be found in the full version of this paper (Sansom & Peyton Jones [1994]).

3.6 Implementing the operational semantics

The state-transition semantics are easily mapped onto a number of different abstract machines, including the G-machine (Augustsson & Johnsson [1989]), the STG-machine (Peyton Jones [1992]) and the TIM (Fairbairn & Wray [1987]), since these abstract machines are all based on the same form of push-enter stack-based model of execution which captures the behaviour of our abstract state-transition semantics. (The particular abstract machines differ in their organisation of lower-level details such as the representation of environments and the update mechanisms employed.)

A flavour of the low-level execution details for our implementation, which is based on the STG-machine, was given in Section 3.1. Observe that the state-transition semantics *only attributes costs to the current cost centre.* This observation justifies the implementation sketched in Section 3.1, in which the consumption of time and space is simply attributed to the current cost centre. Details of our STG-machine implementations, including extended STG-level operational semantics, can be found in Sansom [1994b].

An important aspect of any profiler is the overhead it imposes. Our measurements show that programs take around twice as long to run when they are profiled, which is quite acceptable. If the heap profile is disabled (so that the heap census does not need to be taken), the time increase is reduced to a factor of only 1.7. One reason for this good performance is we profile almost-fully-optimised code, as discussed in Section 3.3.

4 Snares and delusions

It would be nice to report that lexical scoping always does the Right Thing — that is "what the programmer probably expected" — but our experience of using the profiler is otherwise: there are a couple of important cases when it does not. This experience has led us to make the profiler more complicated in order to make it appear simpler and more intuitive to the programmer.

$$cc, \Gamma[x \overset{cc_z}{\mapsto} z] : x \ \Downarrow_{\{cc \mapsto \mathrm{V}\}} \ \Gamma[x \overset{cc_z}{\mapsto} z] : z, \mathrm{SUBCAF}(cc_z, cc) \qquad \textit{Var(whnf)}$$

$$\frac{cc_e, \Gamma : e \ \Downarrow_\theta \ \Delta : z, cc_z}{cc, \Gamma[x \overset{cc_e}{\mapsto} e] : x \ \Downarrow_{\{cc \mapsto \mathrm{V}\} \uplus \theta \uplus \{cc_z \mapsto \mathrm{U}\}} \ \Delta[x \overset{cc_z}{\mapsto} z] : z, \mathrm{SUBCAF}(cc_z, cc)} \ e \not\equiv z \qquad \textit{Var(thunk)}$$

$$\begin{array}{ll} \textbf{where} & \mathrm{SUBCAF}(\texttt{"SUB"}, cc) = cc \\ & \mathrm{SUBCAF}(\texttt{"CAF"}, cc) = cc \\ & \mathrm{SUBCAF}(cc_z \quad, cc) = cc_z \end{array}$$

ccc	Heap	Control	Stack	θ	$\Longrightarrow$	ccc	Heap	Control	Stack	Attribution	rule
cc	$\Gamma[x \overset{cc_z}{\mapsto} z]$	x	S	θ	$\Longrightarrow$	cc_{sub}	$\Gamma[x \overset{cc_z}{\mapsto} z]$	z	S	$\theta \uplus \{cc \mapsto \mathrm{V}\}$	var_1
cc	$\Gamma[x \overset{cc_e}{\mapsto} e]$	x	S	θ	$\Longrightarrow$	cc_e	Γ	e	$(\#x, cc) : S$	$\theta \uplus \{cc \mapsto \mathrm{V}\}$	var_2
cc_z	Γ	z	$(\#x, cc) : S$	θ	$\Longrightarrow$	cc_{sub}	$\Gamma[x \overset{cc_z}{\mapsto} z]$	z	S	$\theta \uplus \{cc_z \mapsto \mathrm{U}\}$	var_3

In the var_1 and var_3 rules $cc_{sub} = \mathrm{SUBCAF}(cc_z, cc)$, where SUBCAF is defined above.
In the var_2 rule $e \not\equiv z$.

Figure 6: Modified Lexical Cost Semantics and State Transition Rules

4.1 Constant applicative forms

Consider the top-level definition:

```
x :: Int
x = if <expensive> then 1 else 2
```

This sort of top-level definition, which has no arguments, is called a *constant applicative form* or *CAF*. When its value is first demanded, its right hand side will be evaluated, and the CAF will be updated with the resulting value. Subsequent demands for its value will incur no cost. Where, though, should the cost of the first evaluation be attributed? If we attach the "SUB" cost centre to the binding, as we do with top-level functions, the one-off evaluation costs would be attributed to the unfortunate cost-centre which happens to first evaluate the CAF. This seems wrong: if the evaluation order changed, then so would the cost attribution, which is most undesirable.

An easy fix is to attribute the one-off evaluation costs to a special "CAF" cost centre which is attached to the CAF bindings in Γ_{init}.[3]

$$\Gamma_{init} \ = \ \{ \ \texttt{x} \overset{\texttt{"CAF"}}{\mapsto} \ \texttt{if <expensive> then 1 else 2}, \dots \}$$

Thus, the cost of evaluating x is attributed to the cost centre "CAF". Alas, this solution still gives unexpected results. Consider the following two definitions:

```
and1,and2 :: [Bool] -> Bool
and1 xs = foldr (&&) True xs
and2    = foldr (&&) True
```

Any red-blooded functional programmer would expect these two definitions to be entirely equivalent: after all, the only difference is an η-reduction. However, they will be given different cost centres in the initial environment Γ_{init} since `and1` is a top level function while `and2` is a CAF:

$$\begin{array}{rcl} \Gamma_{init} & = & \{ \ \texttt{and1} \overset{\texttt{"SUB"}}{\mapsto} \lambda\texttt{xs. foldr (\&\&) True xs}, \\ & & \quad \texttt{and2} \overset{\texttt{"CAF"}}{\mapsto} \texttt{foldr (\&\&) True}, \ \dots \ \} \end{array}$$

Lexical scoping will subsume all the costs of applying `and1` to its call sites, *but the costs of applying* `and2` *will be attributed to the cost centre* "CAF". A major change in cost attribution has resulted from an innocuous change in the program.

The following declaration highlights the problem:

```
y :: Int -> Int
y = if <expensive> then (\a->e1) else (\a->e2)
```

There are two sorts of cost associated with `y`: the one-off costs of deciding which of the two λ-abstractions is to be `y`'s value, and the repeated costs of applying that λ-abstraction. Lexical scoping attributes both costs to the "CAF" cost centre, whereas the programmer would probably expect the costs of applying `y` to be subsumed by its call site. However, the one-off cost of deciding which λ-abstractions is to be applied should still be attributed to the "CAF" cost centre. (This problem only arises under lexical scoping, since evaluation scoping always attributes the cost of applying a λ-abstraction to its call site.)

How, then, can we modify the formal cost semantics to deal with the unexpected attribution of costs to CAFs? Our solution is to extend the notion of subsumed cost to *values* with the "CAF" cost centre, as well as top-level functions with the "SUB" cost centre.

The modified cost semantics are given in Figure 6. In the *Var(whnf)* rule the current cost centre is returned if the closure's cost centre cc_z is "SUB" or "CAF". Previously we only returned the current cost centre for the "SUB" cost centre.

[3] In practice we attach a specific cost centre to each CAF, for example "CAF:x", so that the programmer can separately identify the costs associated with each CAF.

The *Var(thunk)* rule must also be modified to deal with a CAF closure that is not in whnf (cc_e = "CAF"). Since the one-off costs of evaluating a CAF expression to whnf must be attributed to the "CAF" cost centre we always evaluate the bound expression e in the context of cost centre cc_e. However, if the result of that evaluation z is returned with the cost centre q"CAF" (cc_z = "CAF") we return the enclosing cost centre cc instead. The closure is still updated with the result cost centre cc_z so that subsequent references to the closure will be appropriately subsumed by the *Var(whnf)* rule. In this way the costs of repeatedly evaluating the body of a λ-abstraction, declared within the scope of a CAF, are subsumed by the call sites.

The corresponding modifications to the state-transition rules are also given in Figure 6. Apart from the use of the SUBCAF predicate, the main difference is that when evaluation of a thunk is begun, the demanding cost centre, cc, is saved on the stack along with the update marker `#x` (rule var_2). When evaluation of the thunk is complete, the demanding cost centre is restored if the result has the cost centre "CAF" (rule var_3). This mechanism for saving and restoring the current cost centre is very similar to that used in the *case* rules in Figure 5, which save current cost centre before evaluating the scrutinee, and restore it afterwards.

Is the cure worse than the disease? We believe not: before we made this change we received many messages from confused users asking why a large fraction of their program's execution costs were attributed to "CAF". One reason that this fraction is sometimes very high is that our implementation of overloading in Haskell uses *dictionaries*, which are tuples of method functions. It turns out that all of the costs of executing these methods are attributed to the CAF cost centre enclosing the dictionary definition. Indeed, even non-CAF dictionaries can give unexpected cost attribution, precisely because they are introduced by the compiler and invisible to the programmer. The details are in Sansom [1994b], and the solution is similar to that for CAFs.

4.2 CAFs introduced by the compiler

Consider the following definition:

```
f y = let z = exp
      in g z y
```

If `exp` is a constant expression — that is, `y` is not free in `exp` — then the declaration of `z` can be floated to the top-level and turned into a CAF, thus:

```
z = exp
f y = g z y
```

This transformation avoids recomputing `z` every time `f` is evaluated, since it will now only be evaluated once when `f` is first called. However, the programmer is expecting the repeated evaluation of `z` to be subsumed by the caller of `f`, rather than a single evaluation being attributed to a CAF cost centre. Since we want to profile optimised execution, which only evaluates `z` once, we nevertheless allow this optimisation to proceed. The programmer is made aware of any costs attributed to CAFs introduced by the compiler.

The change in cost attribution only arises for constant expressions which are not enclosed by an `scc` annotation. If, instead, the body of `f` were annotated with an `scc`, the `let` floating transformation described in Section 3.3 annotates the floated CAF with an $\texttt{scc}_{sub}$.

```
f y = scc "f" let z = exp
              in g z y
==>
z = scc_sub "f" exp
f y = scc "f" g z y
```

Now the cost of the evaluating `z` is still (correctly) attributed to the cost centre "f" which enclosed the original source declaration.

4.3 Higher order functions

Consider the following definitions:

```
f x y = expensive x y

g1 = scc "g1" (let f' x y = expensive x y
               in h f')
g2 = scc "g2" (h f)

h t = scc "h" (t 2 3)
```

The evaluation of each of `g1` and `g2` gives rise to a single call to `expensive`, one via `f'` and one via `f`. Question: to which cost centres are the costs of evaluating `expensive` attributed? In the first case the answer is clear: the closure `f'` will bear the cost centre "g1", and when `f'` is entered inside `h` the cost centre "g1" will be re-loaded (*Var(whnf)* rule), and the costs of evaluating `f'`'s body will therefore accrue to "g1".

Unfortunately, the situation is different in the case of `g2`, because the argument to `h` is a top-level function, represented in Γ_{init} with cost centre "SUB". When `f` is entered in `h` the current cost centre, "h" is left in place (*Var(whnf)* rule again), and the costs of executing `f`'s body will accrue to "h". However, one would expect the costs of `f` to be subsumed by "g2" since the reference to `f` is in the scope of "g2".

This difference is another example of "unexpected" profiler behaviour. It is easily remedied by the following transformation: *whenever a top-level un-profiled (i.e. subsumed) function is passed as an argument, introduce a* `let`*-binding for that argument.* In this example, we would replace the call `(h f)` with the expression `(let f' = f in h f')`. Now the costs of `f`'s body will accrue to "g2" as one might expect.

5 Using the profiler in practice

A version of the lexical profiler has been distributed with the Glasgow Haskell compiler since Version 0.15 (June 1993) enabling other users to profile their Haskell applications. This has provided us with invaluable feedback about the practical use of the profiler for profiling large applications. The largest applications profiled to date are the Glasgow Haskell Compiler (**ghc**) itself and LOLITA[4] — a natural language system developed by the Artificial Intelligence Research Group at the University of Durham. Each of these applications consist of over 30,000 lines of Haskell code, divided into more than 100 different source modules.

The initial profile of the **ghc** compiler, after explicitly annotating each of the main passes, highlighted two execution hot-spots: the `Renamer` (25%+8% `builtinNames`) and the `Typechecker` (45%) (see Figure 3). Further investigation revealed a compiler transformation bug which duplicated work, and a very inefficient substitution algorithm. Fixing the compiler bug and developing a more efficient substitution algorithm[5], improved the average compilation time by over 50% (Sansom [1994b]).

Profiling of LOLITA initially used automatic annotation to identify the basic functions responsible for a large proportion of the execution time. Improving these functions, which were generally simple operations called hundreds of thousands of times, gave an initial performance improvement of 8%. Further performance improvements were then obtained by developing more efficient algorithms and data structures for system components, such as the word lookup table and the semantic network, which consumed a large proportion of the execution time. The overall improvement to date is about 35% with many further improvements envisaged (Jarvis [1994]).

The experience gained actually using the profiler to profile real programs has provided invaluable feedback. It enabled us to compare the usability of the different cost attribution schemes and it drew our attention to the problem attributing CAF costs under the lexical profiling scheme.

6 Related work

Aside from the profiler developed here, we are aware of only three execution profilers for non-strict functional languages that relate the profiling data back to the program source. The first two profile only space. The third profiles time as well, but has only an interpreted implementation.

The **hbc/lml** heap profiler (Runciman & Wakeling [1993]) was developed concurrently with our heap profiler, and has very similar goals — indeed, we use its post-processor to produce Postscript graphs. Aside from the absence of time profiling, the main difference from our work is that the **hbc/lml** profiler does not provide a mechanism for aggregating information up the call graph. For example, a profile may indicate that heap objects allocated by a certain function, say `map`, occupy a large amount of heap space. However there is no mechanism to determine which application(s) of `map` were responsible for producing these cells (Kozato & Otto [1993]).

More recently the **nhc** heap profiler (Runciman & Röjemo [1994]) extends the **hbc/lml** heap profiling ideas. It provides additional heap profiles which attempt to identify the cause, rather then the presence, of an unexpected space leak. In particular, the *retainer* profile breaks down the heap objects by the (set of) objects that reference them, a most interesting development.

Clack, Clayman & Parrott [1994] have developed a time and space profiler for an interpreted graph reduction system. They report the time and space usage for a set of functions specified by the programmer, the "profiled functions". Though there are encouraging similarities between their profiler and our lexical profiler, there are also some important differences. First, since their system profiles *interpreted* graph reduction, a substantial performance penalty is paid for the benefit of profiling, and the profiling information is not necessarily faithful to a compiled version of the same program.

Second, and more important, if a particular function is called (directly or indirectly) by two or more profiled functions, then it implicitly becomes a separately-profiled function. There is no way to subsume its costs into those of its callers, except by duplicating the function (and the functions it calls...). In contrast, our profiler subsumes the cost of all un-profiled functions even if they are shared. We believe this to be a major strength of our profiling scheme since it enables the costs of the logical "parts" of a program to be aggregated together, regardless of the sharing properties of the program.

It is more straightforward to profile a strict language than a non-strict one, since there are no unevaluated closures to worry about. The semantics in Figure 2 are easily "strictified" by: restricting the *Let* rule to only bind a set of mutually recursive function values; introducing a second *Let* rule which binds the value of a strictly evaluated expression; and omitting the *Var(thunk)* rule. The result is a profiling scheme very similar to the SML profiler described by Appel, Duba & MacQueen [1988]. They introduce the idea of a "current function", to which regular timer interrupts are attributed, and provide a mechanism which enables the costs of "un-profiled functions" to be subsumed by the call site. These two ideas correspond closely to our notions of "current cost centre" and subsumed top-level functions. Their implementation uses side effects to manipulate the current function at the language level; the correct attribution of costs is implicitly maintained by the optimiser's treatment of the operations which manipulate the current function at the function boundaries. Our work could be viewed as a development of the SML profiler, extending it to non-strict languages, and providing a formal model to undergird it.

7 Conclusions

We have described a time and space profiler for a state-of-the-art compiled implementation of Haskell. A key contribution is our formal model for cost attribution. Its main benefit is to provide a precise framework in which to discuss design decisions without becoming enmeshed in the details of a particular implementation. Before we developed the formal model, we found the differences between lexical and evaluation scoping almost impossible to understand and explain clearly, and the modified lexical scheme never occurred to us.

[4]LOLITA: Large-scale, Object-based, Linguistic Interactor, Translator and Analyser.

[5]The new substitution algorithm uses a mutable array (Launchbury [1993b]). It is 50 times more efficient than the original algorithm (Sansom [1994b]).

The formal cost semantics is deliberately slanted towards the programmer rather than the implementation. We also developed a small-step state-transition semantics, which directly expresses the manipulation of cost centres performed by the implementation, and proved it equivalent to the original cost semantics. This proof greatly increases our confidence in the correctness of the implementation.

Our work on the semantics is not yet complete. In particular, we should have a proof that cost attribution is indeed independent of evaluation order; and we should have proofs that a variety of program transformations preserve the cost attribution. It is also somewhat unsatisfactory that we lack a way to be sure that the list of "snares and delusions" in Section 4 is complete.

It is encouraging, though, that the profiler has been successfully applied to a number of large applications. Its use quickly focusses the programmer's attention on the actual execution hot-spots. The profiler has already proved invaluable in evaluating, comparing and tuning the performance of different algorithmic components within large applications.

Bibliography

AW Appel, BF Duba & DB MacQueen [Nov 1988], "Profiling in the presence of optimization and garbage collection," Technical Report CS-TR-197-88, Princeton University.

L Augustsson & T Johnsson [April 1989], "The Chalmers Lazy-ML Compiler," *The Computer Journal* 32, 127–141.

C Clack, S Clayman & D Parrott [March 1994], "Lexical Profiling: Theory and Practice," Dept of Computer Science, University College London, To appear in Journal of Functional Programming.

Jon Fairbairn & Stuart Wray [Sept 1987], "TIM - a simple lazy abstract machine to execute supercombinators," in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, G Kahn, ed., Springer-Verlag, LNCS 274.

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

P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson [May 1992], "Report on the functional programming language Haskell, Version 1.2," *ACM SIGPLAN Notices* 27.

SA Jarvis [April 1994], "Profiling Large Scale Lazy Functional Systems," Artificial Intelligence Research Group, University of Durham.

Y Kozato & GP Otto [June 1993], "Benchmarking real-life image processing programs in lazy functional languages," *Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark.

J Launchbury [Jan 1993a], "A natural semantics for lazy evaluation," *20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina.

J Launchbury [June 1993b], "Lazy imperative programming," *Proceedings of ACM SIGPLAN Workshop on State in Programming Languages*, Copenhagen, Denmark, Available as Research Report YALEU/DCS/RR-968, Yale University.

SL Peyton Jones [April 1992], "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming* 2, 127–202.

SL Peyton Jones, CV Hall, K Hammond, WD Partain & PL Wadler [March 1993], "The Glasgow Haskell compiler: a technical overview," *Joint Framework for Information Technology (JFIT) Technical Conference Digest*, Keele.

C Runciman & N Röjemo [1994], "New dimensions in heap profiling," Departments of Computer Science, Chalmers University and University of York.

C Runciman & D Wakeling [April 1993], "Heap profiling of lazy functional programs," *Journal of Functional Programming* 3, 217–245.

PM Sansom [1994a], "Time profiling a lazy functional compiler," in *Functional Programming, Glasgow 1993*, K Hammond & J O'Donnell, eds., Workshops in Computing, Springer Verlag.

PM Sansom [Sept 1994b], "Execution profiling for non-strict functional languages," PhD thesis, Research Report FP-1994-09, Dept of Computing Science, University of Glasgow, (`ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/tech_reports/FP-94-09_execution-profiling.ps.Z`).

PM Sansom & SL Peyton Jones [Nov 1994], "Time and space profiling for non-strict, higher-order functional languages," Research Report FP-1994-10, Dept of Computing Science, University of Glasgow, (`ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/tech_reports/FP-94-10_timespace-profiling.ps.Z`).

P Sestoft [April 1994], "Deriving a Lazy Abstract Machine," *Dept of Computer Science, Technical University of Denmark*.