J. Functional Programming **8** (3): 201–237, May 1998. Printed in the United Kingdom

© 1998 Cambridge University Press

Profiling large-scale lazy functional programs

R. G. MORGAN

Department of Computer Science, Durham University,
South Road, Durham, UK

S. A. JARVIS

Oxford University Computing Laboratory, Wolfson Building,
Parks Road, Oxford, UK

Abstract

The LOLITA natural language processor is an example of one of the ever-increasing number of large-scale systems written entirely in a functional programming language. The system consists of over 47,000 lines of Haskell code (excluding comments) and is able to perform a wide range of tasks such as semantic and pragmatic analysis of text, information extraction and query analysis. The efficiency of such a system is critical; interactive tasks (such as query analysis) must ensure that the user is not inconvenienced by long pauses, and batch mode tasks (such as information extraction) must ensure that an adequate throughput can be achieved. For the past three years the profiling tools supplied with GHC and HBC have been used to analyse and reason about the complexity of the LOLITA system. There have been good results, however experience has shown that in a large system the profiling life-cycle is often too long to make detailed analysis possible, and the results are often misleading. In response to these problems a profiler has been developed which allows the complete set of program costs to be recorded in so-called cost-centre stacks. These program costs are then analysed using a post-processing tool to allow the developer to explore the costs of the program in ways that are either not possible with existing tools or would require repeated compilations and executions of the program. The modifications to the Glasgow Haskell compiler based on detailed cost semantics and an efficient implementation scheme are discussed. The results of using this new profiling tool in the analysis of a number of Haskell programs are also presented. The overheads of the scheme are discussed and the benefits of this new system are considered. An outline is also given of how this approach can be modified to assist with the tracing and debugging of programs.

Capsule Review

Lazy functional languages provide good control over what is evaluated, but less control over the final time and space behaviour of a program. Profilers provide a partial solution to this problem in that they can be used to detect aberrant run-time behaviour: a full solution relies on the programmer understanding how subsequent modifications to the code can improve its behaviour.

The authors present an extension to the cost-centre profiler which is shipped with the Glasgow Haskell Compiler (GHC). Their new technique records complete stacks of lexical containment for paths in the call-graph: this extends both cost-centre profiling (which only records one level of containment) and lexical profiling (which only records two levels of containment).

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

The technique has the immense advantage that comprehensive run-time information is retained and therefore multiple post-mortem analyses can be run without the need to modify the source code and/or run the program again. However, a key challenge is to implement the technique efficiently. The authors provide very clear and detailed explanations of how efficiency is achieved and present profiling results from large-scale applications.

This excellent paper should be of great interest to anyone who intends to develop performance-critical systems in Haskell using GHC.

---

1 Introduction

One would like to think that most programmers understand what a profiler is by now and that they use a profiler in everyday programming. Even when a program appears to be efficient, an inquisitive programmer will run a program and study the profiling results. In doing so he may find that parts of his code are not as efficient as he hoped, and, as a consequence of this, the code may be changed, recompiled, and profiled once more. It is hoped that the results are an improvement on the original.

A number of reliable profiling tools are now available to Haskell programmers. Of these, the York heap profiler (Runciman and Wakeling, 1993) supplied with the Chalmers Haskell compiler and the Glasgow cost-centre profiler (Sansom and Peyton Jones, 1995) supplied with the Glasgow Haskell compiler are probably the most well-known. Each can measure heap usage during the execution of a Haskell program. At regular intervals during program execution the amount of heap used by each function is recorded; these results can then be viewed as a graph of total heap usage over execution time. The cost-centre profiler also displays time-profiling results for the program. The results of the profiler show, in tabular form, the percentage of the total execution time spent in each of the program's functions.

The tools allow many variations on this theme: the heap profiler can display the results in terms of the producers or constructors; the cost-centre profiler can also display a serial time profile, similar to the heap graphs; there is also the possibility of limiting the number of functions which the programmer profiles, allowing him to concentrate on only that part of the code which he believes to be inefficient.

The aim is to supply the programmer with enough material to identify possible bottlenecks in the program, to identify space leaks, or to locate those parts of the code which use a disproportionate amount of time or memory.

One large programming project which has made considerable use of these profiling tools is the LOLITA natural language processing system at the University of Durham. This system consists of over 47,000 lines of Haskell source code, written in over 180 different modules. The development of the system began in 1986 and there are currently 20 developers working on it (Long and Garigliano, 1994).

LOLITA is an example of a large system which has been developed in a lazy functional language purely because it was felt that this was the most suitable language to use. It is important to note the distinction between this development, where the choice of a lazy functional language is incidental, and projects which are either initiated as experiments in lazy functional languages or have a vested

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

interest in functional languages. Because of this, conclusions drawn regarding the effectiveness of functional languages, and in particular their profilers, were based on genuine end users of functional language technology.

An analysis of the use of the heap and cost-centre profilers within the LOLITA project identified the following problems:

- *Profiling takes a long time* – Compiling and running programs with the profiling options takes longer than without them, due to the extra bookkeeping needed. Programmers also have a tendency to select and reselect functions a number of times before they are completely satisfied with the results. This may require the code to be recompiled and rerun a number of times. For this reason profiling a large program can conceivably take a number of weeks!

- *Profiling results can be misleading* – Once the results of a large program have been produced, the programmer is then faced with the separate problem of interpreting what they mean. Often the programmer will want to display the results at a high level in the code and then decompose them to constituent functions; alternatively, the results may be displayed at a low level in the code and inherited to functions higher in the functional dependency of the program. Previous methods of inheritance have either been inaccurate or limited to a fixed number of generations, because the overheads were considered to be too high to make such a scheme feasible. The heap profiler has no inheritance at all; costs can only be related to the functions or constructors which were directly responsible. The cost-centre profiler does allow a limited form of inheritance in which the cost of functions which are not included in the profile are attributed to their parent functions. However, this only allows information about one particular level in the program to be gathered during any one execution of the program; profiling results may therefore be restricted and problems difficult to identify.

These problems were compounded by the scale of the LOLITA system and it was necessary to design a new method of profiling. This is described in the next section.

2 The cost-centre-stack profiler

When profiling, the programmer is required to identify the portion of the program which he is interested in analysing. The identification of parts of the program may be performed automatically (Runciman and Wakeling, 1993) or more explicitly by the programmer annotating the source-level code (Sansom and Peyton Jones, 1995). It is argued that the latter approach gives the programmer more control over selecting the parts of the code in which he is interested, though automatic annotation can prove to be particularly useful for profiling large programs. Cost-centre profiling makes use of both these techniques, and therefore offers a balanced and flexible

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

approach. It is for this reason and because the cost centre provides time profiling that the cost-centre profiling system is used as the basis for the new profiler.

2.1 Cost centres

A cost centre is described as a label to which costs are assigned. During profiling the programmer may annotate the code with an `scc` expression (set cost centre) which is followed by the cost-centre name. For instance

```
function x =
             scc "costOfFunction"
                  (map expensiveFunction x)
```

will assign to the cost centre `costOfFunction` the costs of the evaluation of the expression `(map expensiveFunction x)`.

In large programs such a scheme could become difficult to use, so to avoid the programmer annotating every function definition, he is also able to select all top-level functions, functions in a named module or just those explicitly added by hand.

When the program is executed any costs incurred are allocated to a single cost centre which is currently in *scope* according to a set of cost-attribution rules. Such rules state that 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 cost centre demands it, excluding, first, the cost of evaluating the free variables of *exp*, and secondly, the cost of evaluating any `scc` expressions within *exp* (or within any function called from *exp*).

This means that any costs incurred by functions within the scope of the enclosing cost centre are aggregated and that the results are not affected by the fact that lazy evaluation may cause the evaluation of the expression within the cost centre to be interleaved with the evaluation of other expressions in the program.

The behaviour of cost aggregation is specified using cost semantics in (Sansom, 1994) which avoids any ambiguity that an informal description may introduce. The costs of evaluating an expression are written in a judgement form:

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

This reads that, in the context of heap Γ and the current cost centre cc, the expression e evaluates to the value z, producing a new heap Δ and a new current cost centre cc_z.

The costs of this evaluation are recorded in θ, a set of mappings between cost centres and costs; $cc \mapsto A$, for instance, would represent the cost of an application charged to the cost centre cc.

The semantics are specified as a series of judgements, statements which will always hold in the system, and a set of rules. The semantic rules are defined as *premises*, the sequents above the line, and the *conclusion*, the sequent below the line. An instance of the application of a logical rule is called an *inference* if it is applied from the

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

premise to the conclusion. Such an instance is called a *reduction* if it is applied in an inverted fashion from the conclusion to the premises. The rule

$$\frac{A_1 \; A_2 \; ... \; A_n}{B}$$

states that if the premises A_1 to A_n hold then the conclusion B also holds; it is possible to apply the rules in either direction.

These rules can be used in the reduction of expressions written in Haskell. Rules exist to cover constructs such as function application, case expressions, variable, let, constructor and scc expressions (Sansom, 1994). The scc expression shows the scoping of cost centres:

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

The *SCC* rule evaluates the expression e to the new expression z in the context of the annotating cost centre cc_{scc} and the heap Γ. The cost reported from e will respect the scope of the scc expression and will be stored in the set of costs θ (no costs are given for the reduction of the scc expression). The resulting heap is shown as Δ and the cost centre as cc_z.

It is the cost of such a reduction, rather than the actual result, which is of primary interest. Reduction sequences of this nature are expressed as proof trees. Rather than showing them in tree form the sequential nature of the reduction can be stressed by setting out the proofs vertically.
So for example, $cc, \Gamma : e \Downarrow_\theta \; \Delta : z, cc_z$ is written:

$1 \;\; cc, \; \{\Gamma\} \; : \; e$
2^x a sub-proof
3^y another sub-proof
$4^z \;\; \{\Delta\} \; : \; z, \; cc_z, \; \theta$

This notation is adopted from the work by Launchbury (1993) and Sansom (1994) with slight differences included to make the reading of the proof easier. For example, each reduction step is numbered so that parts of the proof can be referred to individually. The rules employed in the derivation are indicated by the exponent of each proof step, e.g. 2^x indicates that the second step in the reduction is attained by applying the rule x.

Each derivation may be made up of many smaller derivations. This is demonstrated in lines 2 and 3. At any one point in the derivation a cost centre is currently in *scope* if the costs of the reduction are associated with that particular cost centre. As the reduction proceeds, the current cost centre in scope will change which means that different parts of the program have their evaluation attributed to different cost centres. In the example above, the cost centre cc is in scope for the derivation steps 2 and 3. This simple case is expanded upon.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

An example is considered which specifically makes use of the *SCC* rule.

Working from the conclusion of the rule to the premises (reduction), this rule states that the current cost centre, *cc*, is replaced with the cost centre following the `scc` annotation. So for example, in the code

```
scc "plus" 2 + 3
```

the reduction of the `scc` annotation sets the current cost centre to the new cost centre `plus`. The costs of evaluating the primitive $+$ and the constructors 2 and 3 are also attributed to the cost centre `plus`, as the expression is in the scope of `scc`[1].

The proof of the judgement: cc, Γ: `scc "plus" 2 + 3` $\Downarrow_{\theta''} \Delta{:}5,$ `plus`

1	cc, Γ: `scc "plus" 2 + 3`
2^{scc}	`plus`, $\Gamma{:}(2 + 3)$
3^{prim}	a sub-proof of the primitive $+$ returning $\theta' = \theta \cup \{\texttt{plus} \mapsto P(+)\}$
4^{const}	a sub-proof of the constructors 2 and 3 returning $\theta'' = \theta' \cup \{\texttt{plus} \mapsto (C(2) + C(3))\}$
5^{prim}	Heap $\Delta{:}5,$ `plus` θ''
6^{scc}	$\Delta{:}5,$ `plus` θ''

results in the set θ'' of reduction costs:

$$\{\texttt{plus} \mapsto P(+),\ \texttt{plus} \mapsto (C(2) + C(3))\}$$

These results can be interpreted as being the set of costs which map the cost centre `plus` to, first, the cost of evaluating the primitive $+$, and secondly, the cost of evaluating the the constructors 2 and 3. These costs will be aggregated so that a total cost can be assigned to the cost centre `plus`. This will represent the cost of evaluating all the code defined within the scope of this cost centre.

With more than one cost centre, the allocation of costs to cost centres becomes clear. Consider, for example, the slightly more complicated piece of code:

```
scc "times" 2 * 46 * (scc "plus" 4 + 8 )
```

A reduction of this code will follow a similar form. It is important to notice the point in the reduction at which the cost centre currently in scope changes. In the following reduction this is in lines 7 (when the *SCC* rule is invoked for the second time) to 10 (where the second *SCC* is finally reduced):

[1] The semantic rules for primitives and constructors can be found in Sansom (1994).

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

1 cc, Heap Γ: `scc "times" 2 * 46 * (scc "plus" 4 + 8 )`
2^{scc} `times`, Γ: `2 * 46 * ( scc "plus" 4 + 8 )`
3^{prim} a sub-proof of the primitive $*$
returning $\theta^1 = \theta \cup \{\texttt{times} \mapsto P(*)\}$
4^{const} a sub-proof of the constructors 2 and 46
returning $\theta^2 = \theta^1 \cup \{\texttt{times} \mapsto (C(2) + C(46))\}$
5^{prim} Heap Δ : `92 * ( scc "plus" 4 + 8 )`, `times` θ^2
6^{prim} a sub-proof of the primitive $*$ returning $\theta^3 = \theta^2 \cup \{\texttt{times} \mapsto P(*)\}$
7^{scc} `plus`, Γ: `(92 *) : (4 + 8)`
8^{prim} a sub-proof of the primitive $+$
returning $\theta^4 = \theta^3 \cup \{\texttt{plus} \mapsto P(+)\}$
9^{const} a sub-proof of the constructors 4 and 8
returning $\theta^5 = \theta^4 \cup \{\texttt{plus} \mapsto (C(4) + C(8))\}$
10^{prim} Heap Δ : `92 * 12`, `plus` θ^5
11^{scc} Heap Δ:1104, `times` θ^5
12^{scc} Heap Δ:1104, `times` θ^5

The resulting set θ^5 of reduction costs is:

$$\{\texttt{plus} \mapsto (C(4) + C(8)), \texttt{plus} \mapsto P(+), \texttt{times} \mapsto P(*), \texttt{times} \mapsto (C(2) + C(46)), \texttt{times} \mapsto P(*)\}$$

These semantic rules describe a complete language within which any reduction of a Haskell expression can be demonstrated while describing exactly where the cost of the reduction should go.

Further semantic rules for lambda and constructor expressions, application rules, variable let and case rules, are defined in Sansom's (1994) thesis. Since the semantics for cost-centre stacks only requires modification of the rule for setting cost centres, any further discussion is restricted to this rule.

2.2 Cost-centre stacks

The cost-centre profiler is extended to include the notion of a cost-centre stack (Morgan and Jarvis, 1995). The objective of the cost-centre stacks is to record not just the immediately enclosing cost centre but all the enclosing cost centres to a certain part of a program. This provides a wealth of profiling information that can be used in a variety of ways. For example, with costs attributed to cost-centre stacks, individual cost centres can be removed from the results and their costs can be inherited upwards without any need to modify, recompile and rerun the program. Furthermore, full inheritance profiles can be produced, associating cost centres with the entire cost of reducing the enclosed expression, including the cost of any cost centres contained within that expression.

Many of the failures to effectively produce inheritance profiles have been caused by the schemes used to inherit costs.

Consider an example: 'statistical inheritance' means that the cost of a shared

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

function h would be split between its calling functions, f and g, depending on how many calls were made to the shared function. For example, if there are eight calls from f to h and only two calls from g to h then the time spent in (or below) h will be divided 8:2.

Realistically, these results might not be true: g may have called h twice with a list argument of size 100,000; f may have called h eight times with a list argument of size 10. Statistical inheritance does not account for calls to functions with very different arguments.

A solution to this is to subsume costs. If the shared function h is not profiled then any computation performed in h will be included in the scope of the cost centres f and g. This will accurately distribute the costs between the calling functions. However, the programmer must recognise that this is the case and annotate the functions accordingly, which is not always straightforward in a large program. This method also means that the program must be compiled and profiled once again, which may take some considerable time. Significantly the results will no longer display the costs of h.

An alternative solution is to store the function calls of h from g and h from f (Clack, Clayman and Parrott, 1995). It is possible to assign the costs of h to the two pairs (h, f) and (h, g); the programmer then knows exactly which pair caused the most costs and his attention is drawn to the correct part of the code. Unfortunately if h is relatively inexpensive itself, but calls a function i which is expensive, the above method will no longer work. The pair (i, h) will be attributed large costs while the pairs (h, f) and (h, g) will be attributed small costs, which does not give a reliable indication of how the cost of i should be inherited up to the level of f and g.

The cost-centre-stack profiler extends this idea by replacing pairs of cost centres with a stack of cost centres. Rather than costs being attributed to cost centres, or pairs of cost centres, they are attributed to cost-centre stacks. This allows more accurate results than Sansom's cost-centre profiler and Clack, Clayman and Parrott's lexical profiler; results of which can be replicated using the cost-centre-stack profiler. In recording all the enclosing cost centres to an expression, costs can be unambiguously assigned to those parts of the program which caused them.

The semantics of the cost-centre-stack profiler can be specified with a simple modification to Sansom's cost-centre semantics: In the cost-centre-stack semantics, sequence notation, $\langle x_n, x_{n-1}, .., x_1 \rangle$, is used to represent a stack. Catenation of a cost centre, x_4, to a cost-centre stack, $\langle x_3, x_2, x_1 \rangle$, is performed using $x_4 \frown \langle x_3, x_2, x_1 \rangle$, and results in the sequence $\langle x_4, x_3, x_2, x_1 \rangle$. The cost centre x_4 is said to be at the top of the stack; this corresponds to the current cost centre of Sansom's cost-centre profiler.

The cost-centre-stack extension is modelled in the semantic rules by modifying the reduction rule for the `scc` annotations. It is enough to modify this single rule and use the remainder of the rules in their current form.

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

This rule makes it possible to push the current cost centre onto the top of the

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

cost-centre stack. This shows how the above example then scopes with the new *SCC* rule:

1 cc, Heap Γ: `scc "times" 2 * 46 * (scc "plus" 4 + 8 )`
$2^{scc'}$ $\langle$`times`$\rangle$, Γ: `2 * 46 * ( scc "plus" 4 + 8 )`
3^{prim} a sub-proof of the primitive $*$ returning $\theta^1 = \theta \cup \{\langle \texttt{times} \rangle \mapsto P(*)\}$
4^{const} a sub-proof of the constructors 2 and 46
returning $\theta^2 = \theta^1 \cup \{\langle \texttt{times} \rangle \mapsto (C(2) + C(46))\}$
5^{prim} Heap Δ : `92 * ( scc "plus" 4 + 8 )`, $\langle$`times`$\rangle$ θ^2
6^{prim} a sub-proof of the primitive $*$
returning $\theta^3 = \theta^2 \cup \{\langle \texttt{times} \rangle \mapsto P(*)\}$
$7^{scc'}$ $\langle$`plus, times`$\rangle$, Γ: `(92 *) : (4 + 8)`
8^{prim} a sub-proof of the primitive $+$
returning $\theta^4 = \theta^3 \cup \{\langle \texttt{plus, times} \rangle \mapsto P(+)\}$
9^{const} a sub-proof of the constructors 4 and 8
returning $\theta^5 = \theta^4 \cup \{\langle \texttt{plus, times} \rangle \mapsto (C(4) + C(8))\}$
10^{prim} Heap Δ : `92 * 12`, $\langle$`plus, times`$\rangle$ θ^5
$11^{scc'}$ Heap Δ : `1104`, $\langle$`times`$\rangle$ θ^5
$12^{scc'}$ Heap Δ : `1104`, $\langle$`times`$\rangle$ θ^5

2.3 Secondary semantics for stack inheritance

The cost-centre-stack semantics presented in the previous section would lead to a highly inefficient implementation without any form of optimisation. However, before any optimisations are described it is important to clarify how these cost-centre stacks are used to produce profiling results which are suitable for presentation.

The cost-centre stack information is used to provide the following facilities:

1. Normal cost-centre profiles can be produced. The cost-centre-stack profiler therefore provides at least those facilities offered by the cost-centre profiler.
2. The user may choose to view the profiling results in terms of a subset of the cost centres which were present when the program was compiled and run. Given a particular subset of cost centres, the profile results are the same as those which would be produced by the normal cost-centre profiler if only the subset of cost centres had been present. In effect, the costs attributed to any cost centres not in the selected subset will be moved up to the appropriate enclosing cost centres. This allows a mode of operation in which the program is compiled and run once, with a cost centre on every function, and the user can then obtain a number of different profiling results simply by selecting and de-selecting cost centres.
3. The user can chose between an inheritance profile and a flat profile (or indeed, see a profile which includes both figures). In the inheritance profile, a cost centre is attributed with all of its costs, including those which it incurs through the evaluation of cost centres that it contains.

Producing the above results from the cost-centre stacks is straightforward. A set

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

of selected cost centres is stored in a set $SELECTED$. For a non-inherited profile (case 1 above), this set is equal to the entire set of cost centres, unless otherwise specified by the user.

First consider the non-inherited semantics. Each cost stack in the set θ will contain a selected cost centre nearest the top of the stack. The costs associated with that particular cost stack are attributed to that top-selected cost centre. Consider for instance the stack $\langle c, b, a \rangle$ and the associated costs, say 10. The cost 10 is added to the selected cost centre c. The cost centre c will eventually contain all the costs for the stacks for which c is the top-selected cost centre; it is then printed as a post-processed result. As an example consider the set θ of

$$\{\langle b, a \rangle \mapsto 10, \langle a \rangle \mapsto 20, \langle c, a \rangle \mapsto 10, \langle c, b, a \rangle \mapsto 50\}.$$

Under this scheme the non-inherited post-processed results,

$$a = 20, b = 10, c = 60,$$

would be produced. If the cost centre b was not selected, then the results

$$a = 30, c = 60,$$

would be printed instead. Such a scheme is defined more formally as:

$$\begin{aligned}
&\forall\ cc : cost\ centre;\ \theta : (seq\ cost\ centre) \rightsquigarrow \mathbf{N} \bullet \\
&COST\ cc\ =\ sum\ \{\ cost \mid \forall\ S,\ T : seq\ cost\ centre;\ \exists\ stack : seq\ cost\ centre;\ cost : \mathbf{N} \bullet \\
&\qquad (\{stack \mapsto cost\} \in \theta)\ \wedge\ (S \frown \langle cc \rangle \frown T\ =\ stack) \wedge \\
&\qquad (cc\ \in\ SELECTED) \wedge \\
&\qquad (\forall\ cc' : cost\ centre\ \bullet\ cc'\ in\ S\ \Rightarrow\ cc' \notin SELECTED)\ \}
\end{aligned}$$

The operator $\rightsquigarrow$ represents an injective function and $\mathbf{N}$ represents the set of natural numbers.

The inherited costs are a simple variation on the above. Rather than adding the cost associated with a cost stack to the top-selected cost centre, the cost is added to all the selected cost centres within the cost stack. The results

$$a = 90, b = 60, c = 60,$$

are produced when the above example is considered under this inheritance scheme. Again, a formal definition of the scheme is offered:

$$\begin{aligned}
&\forall\ cc : cost\ centre;\ \theta : (seq\ cost\ centre) \rightsquigarrow \mathbf{N} \bullet \\
&INHERITED_COST\ cc\ =\ sum\ \{\ cost \mid \forall\ S,\ T : seq\ cost\ centre;\ \exists\ stack : \\
&\qquad seq\ cost\ centre;\ cost : \mathbf{N}\ \bullet\ (\{stack \mapsto cost\} \in \theta) \wedge \\
&\qquad (S \frown \langle cc \rangle \frown T\ =\ stack)\ \wedge\ (cc\ \in\ SELECTED)\}
\end{aligned}$$

This ensures that all selected cost centres have their costs inherited.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

2.4 An efficient implementation

Profiling a program written using thousands of functions is potentially an expensive business. The cost-centre information recorded for each function will increase the runtime, which in turn may cause large programs to be almost un-executable.

Introducing cost-centre stacks may therefore seem a limited exercise, as cost stacks require more information to be stored than the original cost-centre model. An efficient implementation must therefore be considered to make such a scheme feasible.

Two efficiency problems were considered key in the implementation of cost-centre stacks; the first was the storage requirements needed for the cost-centre stacks, and the second was the efficiency of the Push operation, required when a new cost centre was entered. The storage problems occur because of the potentially large number and size of distinct stacks to which it might be necessary to attribute costs. The efficiency of the Push operation is critical because it is likely to be called once for every function call in the program. Its task is non-trivial. If the stack which it produces has already been encountered, it must be sure to re-use the space associated with the existing stack rather than allocating further storage.

2.4.1 Space-saving mechanisms

Cost-centre-stack codes:

To produce a practical implementation, applicable to large as well as small programs, cost-centre stacks are replaced with *Cost-Centre-Stack Codes*; each cost-centre-stack code is derived from a *Cost-Centre-Stack Table*. The cost-centre-stack code acts as a pointer to the relevant stack entry in the cost-centre-stack table, so at any one time the cost-centre stack is simply a code which addresses an entry in the cost-centre-stack table. See Fig. 1.

The top left of the figure shows a reference to a cost-centre-stack code at an arbitrary point in a program's execution. This code, 0006, refers to the current cost-centre stack. This code references a point in the cost-stack table, the larger box below. Inside the cost-stack table (top right) a reference to the code 0006 is found; this is the representation of a stack. On the top of this stack is the cost centre F. A back pointer will show the reference to the cost-centre stack onto which F was pushed; this was the stack $\langle A, MAIN\rangle$. Therefore, the stack referred to by the cost-centre-stack code 0006 is $\langle F, A, MAIN\rangle$; this is the current cost-centre stack.

To ensure that stack-table entries are not duplicated, each entry contains an index table of other stacks which can be reached by pushing a single cost centre onto the stack represented by the entry. For example, the stack $\langle MAIN\rangle$ is represented by the entry with code 0001 and contains an index table entry for cost centre A with a corresponding stack pointer 0002. If the cost centre A is entered while the current cost-centre-stack code is 0001, then the Push operation will return cost-centre-stack code 0002, corresponding to the stack $\langle A, MAIN\rangle$.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

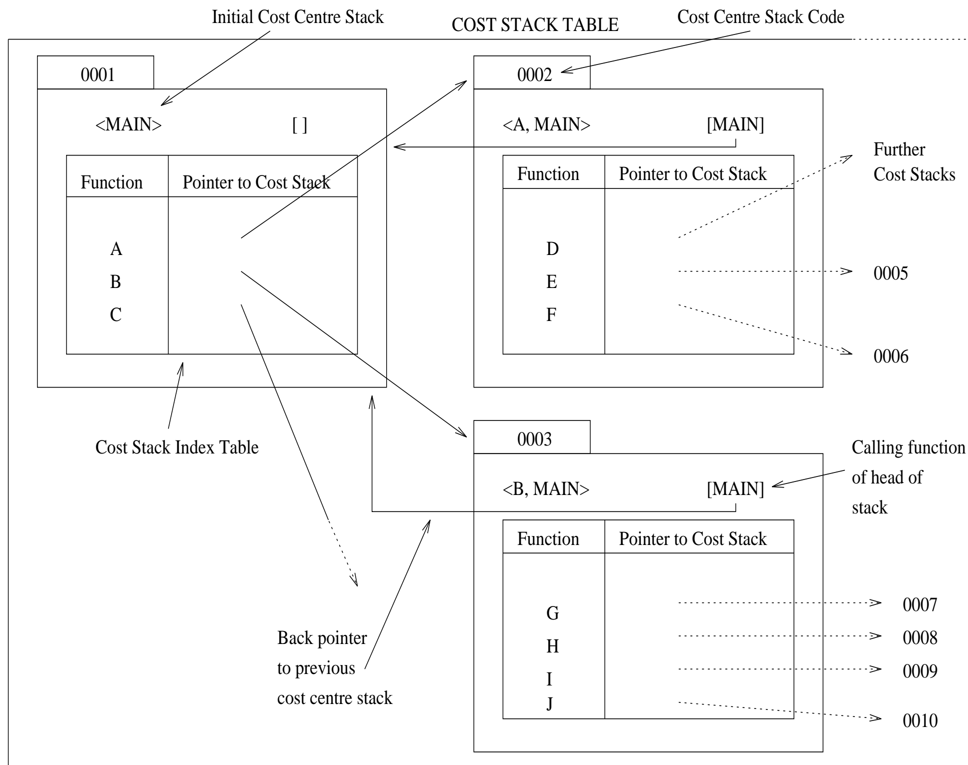


Fig. 1. Implementation of cost stacks.

Compressed stacks

The cost-centre stacks themselves can be compressed. Consider the simple case of a function *a* which calls function *b*, this latter function then calls function *a* again[2]. To avoid this leading to an arbitrarily large number of stacks of the form $\langle a, b, a, b, .. \rangle$, only the top instance of a cost centre is stored in a cost-centre stack. This optimisation has no effect on the profiles produced, because in the flat profile the top-selected cost centre is the only one of interest and in the inherited profile the presence or absence of a selected cost centre on the stack is the only point of interest.

When a function is pushed onto the cost-centre stack, all previous instances of the function are removed from the stack. Such a scheme allows an infinite number

[2] In the case of this and subsequent examples, it is assumed that all top-level functions are annotated with a cost centre.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

of logical stacks to be stored in a single physical stack. The example using two functions a and b requires two such stacks, $\langle a, b\rangle$ and $\langle b, a\rangle$, to represent an infinite number of stacks. Such stacks are referred to as *Compressed Stacks.*

This would seem a good mechanism for keeping the sizes of the stacks small. However, what is the effect on the results? Some figures are given to illustrate the behaviour of this model.

During the duration of the stack $\langle a\rangle$, 3 units of time are accrued. This is written as $\{\langle a\rangle \mapsto 3\}$, representing the cost-centre stack and the costs attributed. When the function b is called, the cost-centre stack $\langle b, a\rangle$ is produced. During this part of the program, 7 units of time are accrued. Finally, the function a is called once again, producing the compressed stack $\langle a, b\rangle$, and a further 1 unit of time is accrued.

This produces the set of cost-centre stacks:

$$\{\langle a\rangle \mapsto 3, \langle b, a\rangle \mapsto 7, \langle a, b\rangle \mapsto 1\}$$

Using the secondary semantics for cost inheritance defined earlier, $COST(a)$ produces the result 4 and $COST(b)$ produces the result 7; these are the non-inherited costs for each of these cost centres. $INHERITED_COST(a)$ produces the inherited result for cost centre a, which is 11; $INHERITED_COST(b)$ produces the result 8.

These results can be compared with the results produced by using uncompressed cost-centre stacks. The function calls above would produce the following set of uncompressed stacks:

$$\{\langle a\rangle \mapsto 3, \langle b, a\rangle \mapsto 7, \langle a, b, a\rangle \mapsto 1\}$$

The difference between the two models is determined by the cost-centre stack $\langle a, b, a\rangle \mapsto 1$. The non-inherited results are the same for this uncompressed stack as they are for the compressed stack; $COST(a)$ is 4, $COST(b)$ is 7. For the inherited results, however, the results are different; $INHERITED_COST(a)$ is 12 and $INHERITED_COST(b)$ is 8. It will be noticed that the inherited costs for a are 1 greater than before. The reason for this is that in the cost-centre stack $\langle a, b, a\rangle \mapsto 1$ the inheritance function adds 1 to cost centre a twice.

Is this really what is required? Should costs be added more than once to a cost centre in the inheritance? If a were in the cost-centre stack 50 times then these inherited results would become 61. This is certainly not desirable.

Using compressed stacks allows mutually-recursive functions to be modelled with a fixed number of stacks. It also allows costs to recursive functions to be inherited once rather than a number of times, producing the inheritance results which are representative of the program's actual costs. The definition of cost inheritance therefore works precisely because of the fact that compressed stacks are used. If uncompressed stacks were used then the definition of inheritance would have to be re-written to prevent the multiple aggregation of single costs.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

2.4.2 The Push operation

The *Push* operation is written *Push(cost centre, cost-stack code)*. When performing a *Push*, the cost-stack code is used to reference the relevant entry in the cost-stack table. The table entry will contain the name of the top cost centre on the stack, a pointer to the previous cost-centre-stack table entry, and a *Cost-Centre Index Table*.

In the remainder of this section cost-centre-stack tables are defined using the following notation:

X^n Represents a cost-centre-stack table entry n with X as the cost centre at the head of the stack;

$X^n \Rightarrow Y^m$ Indicates the cost-centre-stack index table entry from cost-centre stack n to cost-centre stack m, given a push of Y onto n;

$X^n \leftarrow Y^m$ Indicates a back pointer from cost-centre stack m to cost-centre stack n.

The cost-centre stack is represented simply as the head of the stack, as the remainder of the stack can be accessed by the back pointer.

Initially, a program starts with a cost-centre-stack table containing only one entry, $MAIN^1$, further table entries are then added by the Push operation. There are a number of cases which need to be considered in the construction of cost-centre stacks. Each individual case will help to demonstrate that the choice of structure is key to the implementation of efficient stacks and how neatly the scheme can be constructed.

Consider the cost-centre-stack table represented by the following:

$$\begin{array}{c} MAIN^1 \overset{\leftarrow}{\Rightarrow} A^2 \overset{\leftarrow}{\Rightarrow} B^3 \\ \Downarrow\uparrow \\ C^4 \end{array}$$

The cost-centre stack which contains the cost centre $MAIN$ is represented by the cost-centre-stack code 1; the cost-centre stack referred to by the code 4 is, $\langle C, A, MAIN \rangle$; the back pointer from this stack points to the cost-centre stack $\langle A, MAIN \rangle$.

Some examples of the push operation using this cost-centre-stack table can now be considered in turn.

Case 1

The first case considers the situation in which a cost centre is being pushed onto a stack which has already had that cost centre pushed onto it before. For instance, in the cost-stack table shown above, pushing the cost centre B onto the cost stack represented by the cost-stack code 2, results in the cost-stack code 3.

The implementation of this case is efficient. The cost-centre index table for stack 2 will contain entries for cost centres B and C, so all that is required of the Push op-

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

eration is to scan the index table and return the stack code found. Even a simplistic algorithm can perform this scan in $O(n)$ time where n is the number of entries in the table. Since n is bounded by the number of functions called from the function whose cost centre is at the head of the stack, it is typically small and does not usually get larger with overall program size. An $O(n)$ scan has been found to be an adequate result.

Case 2

In the second case, the cost-stack index table would have no reference to that particular cost centre (there will have been no previous attempt to push this cost centre onto the current cost stack) and the cost centre would not have appeared anywhere in the cost stack. In this situation, it is enough to add the new cost centre to the index table of the current stack table entry and thus create a reference to a new cost stack.

In the previous cost-centre-stack table, pushing the cost centre C onto the cost-centre stack with the code number 3 results in the following cost-centre-stack table and the cost-centre-stack code 5.

$$\begin{array}{ccccccc} MAIN^1 & \overset{\leftarrow}{\Rightarrow} & A^2 & \overset{\leftarrow}{\Rightarrow} & B^3 & \overset{\leftarrow}{\Rightarrow} & C^5 \\ & & \Downarrow\uparrow & & & & \\ & & C^4 & & & & \end{array}$$

The back pointers are used during this case to check to see if the cost centre C already exists in the cost-centre stack 3. The overheads of this case are therefore $O(n+m)$, where n is the number of memoised entries contained in the cost-centre-stack index table of stack 3 and m is the depth of the cost-centre stack.

Case 3

The third case considers a cost centre already existing in a current stack.

<u>First subcase</u>

The first subcase which is considered demonstrates that stacks can be built without any intermediate stacks having to be created.

Consider the case where the cost centre B is pushed onto the cost-centre stack identified by the code 5 in the following cost-centre-stack table.

$$\begin{array}{ccccccc} MAIN^1 & \overset{\leftarrow}{\Rightarrow} & A^2 & \overset{\leftarrow}{\Rightarrow} & B^3 & \overset{\leftarrow}{\Rightarrow} & C^5 \\ & & \Downarrow\uparrow & & & & \\ & & C^4 & & & & \end{array}$$

The resulting cost-centre-stack table will, perhaps surprisingly, be as follows:

$$\begin{array}{ccccccc} MAIN^1 & \overset{\leftarrow}{\Rightarrow} & A^2 & \overset{\leftarrow}{\Rightarrow} & B^3 & \overset{\leftarrow}{\Rightarrow} & C^5 \\ & & \Downarrow\uparrow & & & & \Downarrow \\ & & C^4 & & \overset{\leftarrow}{\Rightarrow} & & B^6 \end{array}$$

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

The resulting cost-centre-stack code is 6. The subtlety lies in the fact that from the cost-centre-stack code 5 a new cost-centre stack is built, 6, with B as the top cost-centre stack. The duplicate copies of the cost centre B are avoided in this cost-centre stack by the fact that the back pointer from cost-centre stack 6 points to 4 and not 5. This results in the stack $\langle B, C, A, MAIN \rangle$ and not $\langle B, C, B, A, MAIN \rangle$, results which were expected.

Second subcase

A second subcase is introduced with the following example: The cost centre B is pushed onto the cost-centre stack 4 in the following cost-centre-stack table:

$$MAIN^1 \overset{\leftarrow}{\Rightarrow} A^2 \overset{\leftarrow}{\Rightarrow} B^3 \overset{\leftarrow}{\Rightarrow} C^4$$

The cost centre B already exists on the cost-centre stack. According to the rules introduced in the compressed-stack scheme, only one instance of the cost centre can appear in the cost-centre stack at any one time.

The previous stack pointers make such an operation possible. They allow pre-existing stacks to be located and new stacks to be created if necessary. It is now feasible to follow the previous stack pointers back and check to see if the cost centre B had already been pushed onto the stack. (This checking process will also have taken place in cases 1 and 2.)

Once the previous reference to B has been identified, the new stack is built with the previous occurrence of the cost centre effectively removed from the stack. This results in the cost-centre stack 6 in the following cost-centre-stack table:

$$\begin{array}{ccccccc} MAIN^1 & \overset{\leftarrow}{\Rightarrow} & A^2 & \overset{\leftarrow}{\Rightarrow} & B^3 & \overset{\leftarrow}{\Rightarrow} & C^4 \\ & & \Downarrow\uparrow & & & & \Downarrow \\ & & C^5 & & \overset{\leftarrow}{\Rightarrow} & & B^6 \end{array}$$

Again, from cost-centre stack 4 the cost centre B is added without a back pointer to the cost-centre stack 4. Instead it points to a cost-centre stack 5, an intermediate stack which needed to be built to support the new cost-centre-stack table. The cost-centre stack 6 produced in this case is $\langle B, C, A, MAIN \rangle$; the intermediate stack which is built is $\langle C, A, MAIN \rangle$.

It is possible that this intermediate stack will have needed to be built at some previous stage in the program's execution. If so, then the stack will already exist, as in case 1 above. Of course it might be possible that such a stack will not need to be built. Intermediate stacks can therefore be constructed with reduced storage, without explicit profiling details such as time and heap usage.

Third subcase

It is possible that once the occurrence of the new cost centre in the cost-centre stack has been identified, the resulting new stack already exists[3]. A reference to this stack is produced as the result.

[3] This is a slightly more complicated example of case 1.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

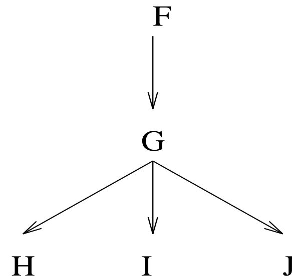
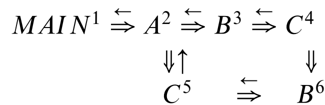
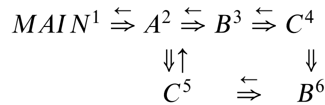


Fig. 2. Example call-graph.

Extending the example seen above,



and pushing the cost centre C onto the cost-centre stack 6, produces the cost-centre-stack code 4 in the following cost-centre-stack table:



From the cost-centre stack $\langle B, C, A, MAIN \rangle$ the function C is called again. The crude approach to creating the new cost centre would have been to remove C from the stack and build the stack $\langle C, B, A, MAIN \rangle$ again, but this is not necessary as this stack already exists. The previous stack pointers allow the location of the stacks $\langle A, MAIN \rangle$, $\langle B, A, MAIN \rangle$, and finally the cost-centre stack $\langle C, B, A, MAIN \rangle$, to be identified, all of which exist in this case.

The previous stack pointers allow 'the stack which came before' to be identified. They also allow previous occurrences of cost centres to be found, and found in an economic manner. The backtracking procedure which has been described only needs to be followed until the previous occurrence of a cost centre is found; the cost centre may be discovered through the first previous stack pointer (in a recursive call to a function), or after 50 previous stack pointers, although it is certain that if the previous stack pointers are exhausted and the first cost centre is found then the cost centre has not been pushed before. Most of the time the backtracking will not need to go all the way back to the first cost centre, as the previous stack pointers ensure that the search is done as efficiently as possible.

If n is the number of cost centres in the current cost-centre stack and m is the total number of entries in the cost-centre index tables for all n_i, then the complexity of this last case is $O(n+m)$.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

2.5 Two examples

Consider the call-graph in Fig. 2. The current cost-centre stack, after calls from F to G and then successful calls from G to H and I, would simply be $\langle G, F \rangle$ at some point in time when I had been executed and G was still executing. The corresponding index table to this cost-centre stack would have references to the functions H and I and pointers to their corresponding cost-stack codes.

The cost-stack code 2 may represent this cost-centre stack in the cost-centre-stack table:

$$\begin{array}{c} F^1 \overset{\leftarrow}{\Rightarrow} G^2 \overset{\leftarrow}{\Rightarrow} H^3 \\ \Downarrow\uparrow \\ I^4 \end{array}$$

The cost-stack table will also include references to the cost-stack codes 3 and 4.

Cost-centre-stack code 3 represents the cost-centre stack $\langle H, G, F \rangle$
Cost-centre-stack code 4 represents the cost-centre stack $\langle I, G, F \rangle$

These codes, plus the cost-stack code 1 for the cost-centre stack $\langle F \rangle$, make up the cost-stack table which will be referred to as T.

On calling function J, the operation $Push(J, 2)$, the cost-centre-stack system will:

1. Look up the cost-stack code 2 in the cost-stack table and return the state.
2. Determine whether its index table has a reference to the cost centre.
3. If there is a reference to the cost centre, then the pointer to the corresponding cost-stack code is returned as the new cost-centre stack.
4. If not, the previous stack pointers are followed back to determine whether there is a previous reference to the cost centre in the cost-centre stack. The code of each cost-centre stack visited is recorded on a conventional stack until the previous reference to the cost centre is identified or the top cost-centre stack, $\langle MAIN \rangle$, is reached. If the pushed cost centre is identified, its parent state is found; from here cost centres are popped off the conventional stack and the appropriate index-table entry is followed until no more cost centres remain on the conventional stack. At this point the current stack will be the stack which is required as the result of the Push operation, and the only remaining action is to set up an appropriate index table entry in the stack that was originally pushed. If any of the index-table entries are not present for a cost centre from the conventional stack, then a new cost-centre-stack table entry must be created and an appropriate entry added to the index table. The same thing will then need to be done for each of the remaining entries in the conventional stack.
5. If there are no previous references to the cost centre in the cost-centre stack then a new table entry is created. This is linked to the stack, given as an argument to Push, via a back pointer and an index-table entry.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

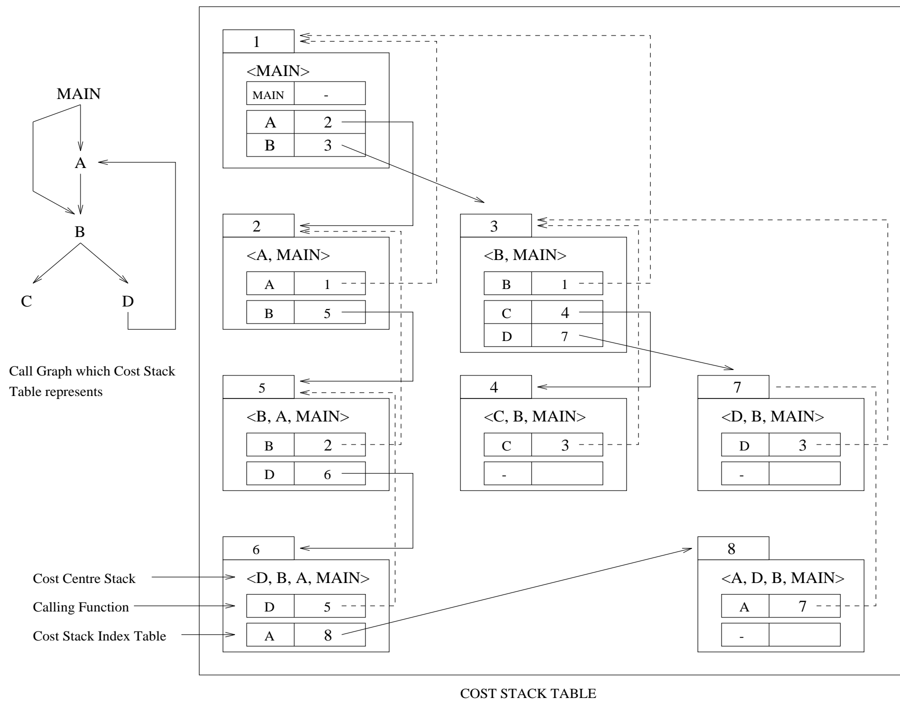


Fig. 3. Example cost-centre-stack table.

The resulting situation in the example is:

$$\begin{array}{c} J^4 \\ \Uparrow\downarrow \\ F^1 \overset{\leftarrow}{\Rightarrow} G^2 \overset{\leftarrow}{\Rightarrow} H^3 \\ \Downarrow\uparrow \\ I^4 \end{array}$$

The example considers function calls in a simple situation. It is more likely that a call-graph of a program execution is far more complicated, with calls to previously-called functions and recursive definitions. The cost-stack implementation responds equally well in such situations. Consider the call graph and cost-stack table in Fig. 3.

An interesting case is presented when function D calls function A which is already in the current cost stack, state 6 in the example cost-stack table. The cost-stack rules state that a function can only appear in the cost stack once. It is therefore necessary to delete the previous reference to function A and create a new cost stack.

The back pointers are followed, popping each cost-centre-stack code onto a conventional stack, until cost centre A is found.

From the current cost stack indexed by the number 6 in Fig. 3, the back pointers are followed starting with the current function at the head of the stack D. The descending stack $\langle B, D\rangle$ is recorded before the previous call to A is found. This previous reference to A is not added to the descending stack and its parent function

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

$MAIN$ is found. From the $MAIN$ function the descending stack $\langle B, D \rangle$ is unwound to create the new cost stack. It is quite possible that part, or all, of the new cost stack already exists. For instance in the example the function B has already been called from the function $MAIN$ and this cost stack already exists. Likewise, the function D has already been called from the function B so there is no need to create a new cost stack $\langle D, B, MAIN \rangle$. Finally, the function call to A is added to the cost stack. This creates state 8 in the figure.

There will in general be many cases when the cost-centre stack which is referenced already exists and it is only the exception when a new stack must be created. For most of the time, therefore, updating the current cost-centre stack simply means using a reference to another cost-centre stack in the cost-centre-stack index table; the expense is simply the look-up in the cost-centre table, an operation which is easily optimised.

3 Results of the profiler and post-processor

The cost-centre-stack profiler, outlined in the preceding section, has been implemented as a modification to the Glasgow Haskell Compiler version 0.22. The cost-stack code is written in C and is included within the GHC run-time code. A large amount of effort has been spent on making this code efficient and providing tools for analysing the information produced by the profiler.

In this section results from the cost-centre-stack profiler and the accompanying post-processor are presented (Jarvis and Morgan, 1996). Case studies are used to compare the results of the cost-centre profiler and the cost-centre-stack profiler. The results of both large and small programs are discussed.

The first example demonstrates the effect which the cost-centre-stack profiler has on the results of shared functions. It is difficult to illustrate the benefits of accurate cost inheritance when working with larger programs; the quantity of code means that it is not always easy to see why the results are so different. Therefore, the first example program is only 14 lines long. The results gained from the cost-centre profiler and the cost-centre-stack profiler are significantly different. This first example is also used to explain the post-processing procedure.

Results are then presented from the LOLITA system. This is a considerable example and contains hundreds of thousands of function calls. It is therefore a substantial test case for the profiler. Attention is paid to the process of profiling using the inheritance results; this contains some different observations from those identified in previous methods of profiling.

The final set of results are from the `nofib` benchmark suite, including the traditional benchmark program `clausify`. From these results some general conclusions are drawn.

3.1 Introductory example

The first set of results are from a simple program designed to be computationally expensive. The program, which repeatedly reverses lists of numbers, makes use of a number of shared functions. These functions illustrate the difference in the results

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

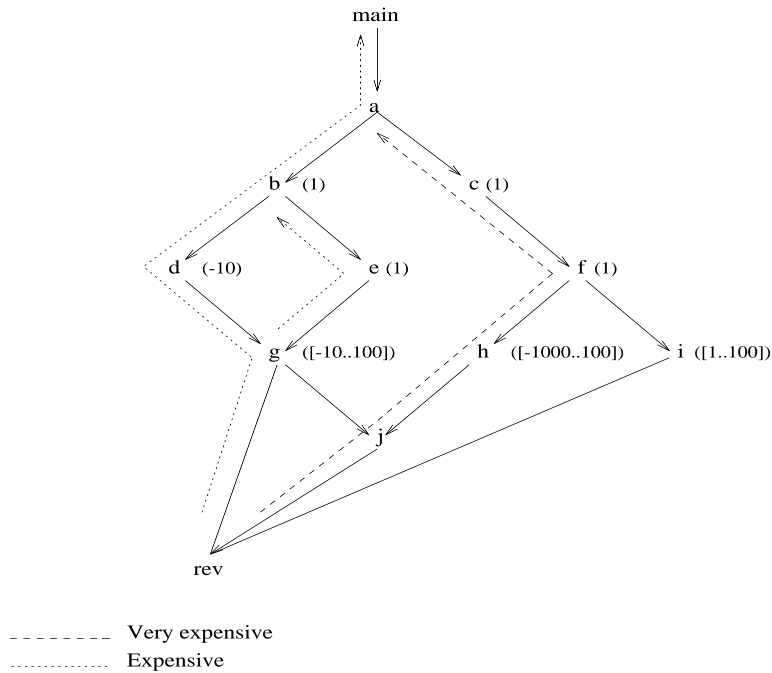


Fig. 4. Call-graph of the experimental program.

achieved using a method of cost inheritance, produced by the cost-centre-stack profiler, and a method of flat profiling, produced by the original cost-centre profiler. The example is clearly contrived, but serves to illustrate the basic differences between the two sets of profiling results. The program

```
module Main where
main = print (length a)
a = (b 1) ++ (c 1)
b x = (d x) ++ (e x)
c x = f x
d x = g (x - 10)
e x = g x
f x = (h x) ++ (i x)
g x = (j [x..100]) ++ (rev (rev (rev (rev [x..100]))))
h x = j [-1000..100]
i x = rev (rev (rev (rev [x..100])))
j l = rev (rev (rev (rev (rev (rev (rev l))))))

rev [] = []                -- An O(n^2) reverse function
rev (x:xs) = (rev xs) ++ [x]
```

is depicted in the call-graph in Fig. 4. The shared functions g and j serve to illustrate expensive functions, if called with suitably large arguments. It is a function call from h to j which causes the largest amount of computation; the function call from g to

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

```
Thu Jun 20 14:46 1996 Time and Allocation Profiling Report  (Final)
                   (Hybrid Scheme)

           run +RTS -pT -H60M -RTS

COST CENTRE      MODULE      GROUP          scc subcc  %time %alloc
Main_rev         Main        Main         20867 20834   99.7   99.8
Main_f           Main        Main             3     2    0.1    0.0
Main_a           Main        Main             2     3    0.0    0.0
Main_b           Main        Main             2     2    0.0    0.0
Main_g           Main        Main             2    12    0.0    0.0
MAIN             MAIN        MAIN             1     0    0.0    0.0
Main_c           Main        Main             1     3    0.0    0.0
Main_d           Main        Main             2     1    0.0    0.0
Main_e           Main        Main             1     1    0.0    0.0
Main_h           Main        Main             1     2    0.0    0.0
Main_i           Main        Main             1     4    0.0    0.0
Main_j           Main        Main             6    21    0.0    0.0
PRELUDE          Prelude     Prelude          0     0    0.2    0.1
Main_main_CAF    Main        Main             0     0    0.0    0.0
CAF.Main         Main        Main             0     3    0.0    0.0
Main_h_CAF       Main        Main             0     0    0.0    0.0
Main_g_CAF       Main        Main             0     0    0.0    0.0
Main_i_CAF       Main        Main             0     0    0.0    0.0
```

Fig. 5. Results of the cost-centre profiler.

j causes significantly less. The arguments passed to the called function are shown in brackets in the figure; for example function e calls function g with the argument 1.

The program is time-profiled with the cost-centre profiler[4] and the results are displayed in Fig. 5. As expected, the reverse function rev (from the program Main) accounts for nearly all of the execution time, 99.7% in total[5]. The remaining 0.3% of costs are attributed to the functions f (0.1%) and the prelude (0.2%). This last figure is due to the catenation function (++) used throughout the program.

The flat cost-centre profile presented in Fig. 5 does not provide the programmer with very useful information. It is not clear which of the functions g, j and i, which share the function calls to the utility function rev, is responsible for the highest proportion of the costs. Without any recompilation and reprofiling further results are impossible to calculate.

The cost-centre-stack profiler produces two sets of results. Firstly, a flat profile is produced in the same way as for the cost-centre profiler, Fig. 6. It is important to note that the results of the cost-centre-stack flat profile are almost identical to the results of the flat profile of the cost-centre profiler. This is an important observation; it shows that the results of the original cost-centre profile will not be distorted by

[4] Compile-time flags: -prof -auto-all; Run-time flags: -pT.

[5] This $O(n^2)$ reverse function is included in the Main program to prevent costs from being assigned to the $O(n)$ reverse function defined in the PRELUDE library.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

```
Total Number of Time Ticks = 1237

Cost centre   Ticks as head of CC stack   Total time ticks   %time

Main_rev      1237                        1237               100.0
Main_main     0                           0                    0.0
Main_a        0                           0                    0.0
Main_b        0                           0                    0.0
Main_c        0                           0                    0.0
Main_d        0                           0                    0.0
Main_e        0                           0                    0.0
Main_f        0                           0                    0.0
Main_g        0                           0                    0.0
Main_h        0                           0                    0.0
Main_i        0                           0                    0.0
Main_j        0                           0                    0.0
Prelude       0                           0                    0.0
```

Fig. 6. Flat profile of the cost-centre-stack profiler.

```
<Main_rev,Main_j,Main_h,Main_f,Main_c,Main_a,Main_main,> 1181 TICKs
<Main_j,Main_h,Main_f,Main_c,Main_a,Main_main,>             0 TICKs
<Main_f,Main_c,Main_a,Main_main,>                           0 TICKs
<Main_j,Main_g,Main_e,Main_b,Main_a,Main_main,>             0 TICKs
<Main_rev,Main_j,Main_g,Main_e,Main_b,Main_a,Main_main,>   16 TICKs
<Main_g,Main_e,Main_b,Main_a,Main_main,>                    0 TICKs
<Main_rev,Main_g,Main_e,Main_b,Main_a,Main_main,>          10 TICKs
<Main_a,Main_main,>                                         0 TICKs
<Main_b,Main_a,Main_main,>                                  0 TICKs
<Main_i,Main_f,Main_c,Main_a,Main_main,>                    0 TICKs
<Main_rev,Main_i,Main_f,Main_c,Main_a,Main_main,>           7 TICKs
<Main_main,>                                                0 TICKs
<Main_c,Main_a,Main_main,>                                  0 TICKs
<Main_g,Main_d,Main_b,Main_a,Main_main,>                    0 TICKs
<Main_rev,Main_g,Main_d,Main_b,Main_a,Main_main,>          11 TICKs
<Main_j,Main_g,Main_d,Main_b,Main_a,Main_main,>             0 TICKs
<Main_rev,Main_j,Main_g,Main_d,Main_b,Main_a,Main_main,>   12 TICKs
<Main_d,Main_b,Main_a,Main_main,>                           0 TICKs
<Main_e,Main_b,Main_a,Main_main,>                           0 TICKs
<Main_h,Main_f,Main_c,Main_a,Main_main,>                    0 TICKs
Prelude no stack
```

Fig. 7. Results of the cost-centre-stack profiler.

the inclusion of the cost-centre stacks in the compiler. The 0.3% error is due to sampling differences.

The cost-centre-stack profiler also produces the cost-centre stacks found in Fig. 7. Each cost-centre stack is displayed with the units of time spent computing values within its scope. These results are useful as they immediately indicate the cost stack

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

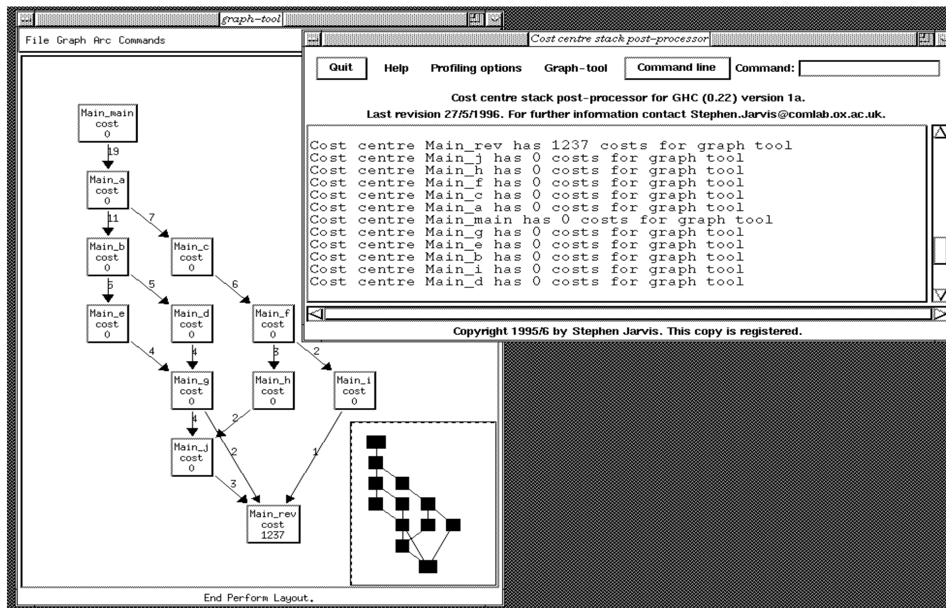


Fig. 8. Post-processor displaying non-inherited post-processed results.

of the most computationally expensive part of the program. As expected, function `rev` is at the head of this stack:

```
<Main_rev,Main_j,Main_h,Main_f,Main_c,Main_a,Main_main> 1181 TICKs
```

The reader will observe from Fig. 7 that the function `rev` is also at the head of five further cost-centre stacks which show the path of cost centres to the function `rev` via `d`, `e` and `i`. The programmer is presented with a complete set of unambiguous results which avoids there being any misunderstanding when they are interpreted.

The first stage of post-processing involves the cost-centre stacks being transformed, using a C program, into a format which can be interpreted by a graph-tool. Cost centres in the stacks become graph *nodes* and adjacent cost centres in a cost-centre stack are connected with directed *arcs*.

The total number of time ticks is calculated for each of the functions at the head of each cost-centre stack. This figure is divided by the total number of time ticks recorded to obtain a percentage. This is equivalent to calculating a flat profile, as in Fig. 6.

The second stage of post-processing involves the programmer selecting those functions which he is interested in profiling. This activity has been moved from pre-profiling to post-profiling. For the sake of this initial example all functions are selected.

This task is implemented in a second C program, taking the graph-tool input file and producing an augmented input file depending on which cost centres are selected. The resulting file is then loaded into the graph-tool; this is the third stage of post-processing.

The structure of the program becomes clear when the results are displayed in the graph-tool. In this example the programmer is presented with the call-graph of the program containing all top-level functions. These results are shown in Fig. 8.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

```
Total Number of Time Ticks = 1237

Cost centre  Ticks in CC stack              Total time ticks  %time

Main_main    1181 + 16 + 10 + 7 + 11 + 12   1237              100.0
Main_a       1181 + 16 + 10 + 7 + 11 + 12   1237              100.0
Main_b       16 + 10 + 11 + 12                49                4.0
Main_c       1181 + 7                       1188               96.0
Main_d       11 + 12                          23                1.9
Main_e       16 + 10                          26                2.1
Main_f       1181 + 7                       1188               96.0
Main_g       16 + 10 + 11 + 12                49                4.0
Main_h       1181                           1181               95.5
Main_i       7                                 7                0.6
Main_j       1181 + 16 + 12                 1209               97.7
Main_rev     1181 + 16 + 10 + 7 + 11 + 12   1237              100.0

Prelude
```

Fig. 9. Inherited results of the cost-centre-stack profiler.

Each node in the graph contains the cost-centre name and the associated costs (in time ticks or as a percentage; the figure shows the former). Each arc in the call-graph is annotated with a number. This number indicates in how many cost-centre stacks this arc was found.

The results are also displayed textually in the cost-stack post-processor. The programmer can view the cost-centre stacks, select cost centres or choose different profiling options from this window. All functions are executed within a couple of seconds without any further execution or compilation of the program.

The post-processing tool is also able to perform inheritance of results, accurately inheriting the profiling results to all the selected functions. This is achieved by adding the costs associated with each cost-centre stack to every cost centre in the cost-centre stack. This mechanism is demonstrated in Fig. 9.

These results can also be displayed by reloading the graph-tool with the new input file, see Fig. 10. If it was not already clear in the previous results, the expensive arm of the graph now becomes immediately obvious. To emphasise this fact, it is possible to highlight the expensive arm of the graph (or display this arm of the graph only). This proves to be a useful function in graphs which contain a large number of nodes.

There are two issues which must be addressed in the analysis of these results. The first is the *usefulness* of the cost-centre-stack data and the post-processing techniques for presenting these data. The second is that of the *overheads* involved in collecting the extra data.

3.1.1 Usefulness

The cost-centre-stack information allows a dynamic call-graph of the program (such as those seen in Fig. 8 and in Fig. 10) to be displayed. Even if the programmer

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

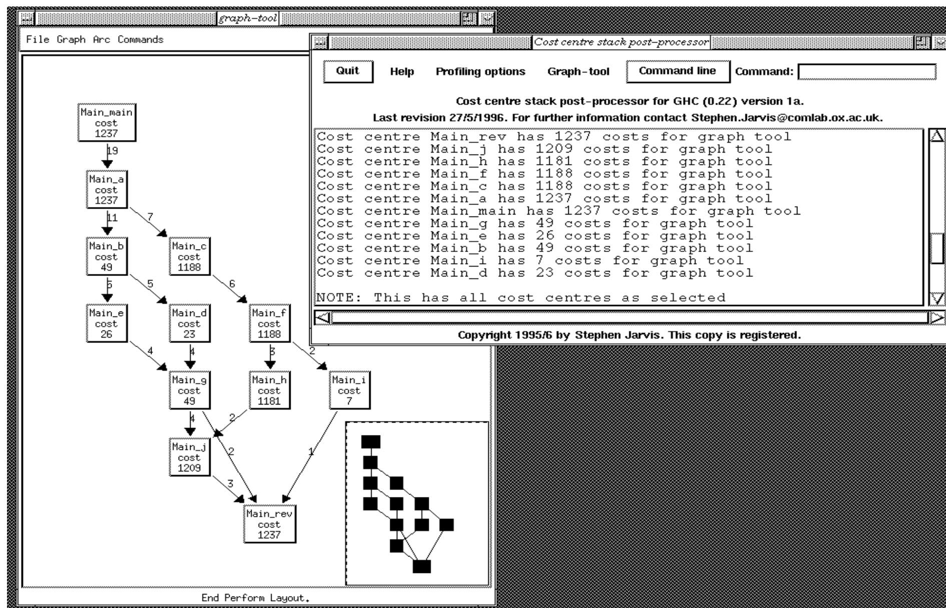


Fig. 10. Post-processor displaying inherited post-processed results.

is familiar with the code this makes the task of determining the relationships between parts of the code easier, particularly in the context of a large program. This information has not previously been shown in the profile of a program.

The dynamic call-graph constructed from compressed cost-centre stacks may look different to a call-graph produced by static analysis of the program, particularly if the program contains many mutually-recursive function definitions.

Compressed stacks ensure that cost centres will only appear in a cost-centre stack once, therefore removing the possibility of loops from the final call-graph. This may appear to the programmer to be different to the way in which the program is actually coded. It does not, however, affect the accuracy of the profiling results and evidence has shown that the programmer is quickly able to adjust to the way in which the results are displayed. It is thought that the simplified call-graphs may even make it easier for the programmer to identify expensive portions of the code.

Profiling with cost-centre stacks allows the complete set of program costs to be recorded. They are an accurate record of the program's computational behaviour, and therefore a true profile of the program in the sense that no statistical averaging has been used to produce the results.

The post-processor allows these results to be explored instantaneously and without any further execution or compilation of the program. This has not previously been possible when profiling a program. Fig. 10 shows the complete set of inherited results. Post-processing enables the expensive portion of the code to be identified, this may be highlighted in the call-graph. The programmer is then free to identify the cause.

There are a number of profiling and graph-tool options available to the programmer which allow, amongst other things, the most expensive arm of the graph to be

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

displayed, functions to be selected, and flat and inheritance profiles of the program to be produced. None of these options take more than two or three seconds to execute.

The graph in Fig. 10 clearly shows the distribution of costs in the program, focusing the programmers attention on the functions `c`, `f`, `h` and `j`. The programmer can quickly identify the function call from `h` to `j` as causing a significant amount of computation.

Using the post-processor it is also possible to select and de-select cost centres. In the example the programmer might have chosen to view the flat profile with the cost centre `Main_rev` de-selected. The post-processor would accurately subsume the costs of `Main_rev` to its calling functions, again highlighting `Main_j` as part of the expensive arm of the program. Repeating this exercise on `Main_j` would confirm that the call from `Main_h` to `Main_j` was instigating most of the program costs.

A similar top-down (as opposed to bottom-up) approach to profiling can be performed using the post-processor. This is easy to achieve without reprofiling and recompiling the program and the time benefits are considerable.

3.1.2 Overheads

The study of the overheads for the cost-centre-stack profiling scheme is important. Previous profiling literature has shown that earlier attempts at similar cost collection methods were abandoned because of the extremely high overheads. The success of the cost-centre-stack method of recording results relies on the fact that the overheads are low enough to make such a system practical.

There are a number of overheads to consider:

- The size of the cost-centre-stack table should not become so large that cost-centre-stack profiling becomes impossible on normal workstations.
- The time that the program takes to compile and any extra heap space needed during compilation should be acceptable. That is, the extra compilation overheads should be small enough to make cost-centre-stack profiling preferable to repeated compilations of a cost-centre profile. Although most of the changes to the Glasgow Haskell compiler are to the run-time system, the changes to the compiler optimiser and the generation of extra run-time code mean that the size of the executable file is slightly larger.
- Finally, the execution time costs should be small enough to make the collection of costs practical. The run-time overheads should not be unacceptably high and the extra heap needed for execution should not be unacceptably large.

These overheads are considered for the first example program:

Cost-centre stacks: execution of the first example program produces 20 cost-centre stacks (when using the `-prof -auto-all` compile time option). There are a total of 12 cost centres in this program.

Analysing the structure of the cost-centre stacks and calculating how many bytes are needed to store this information shows that the size of the cost-centre-stack table

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

for this program is 528 bytes. Compared with the total size of the executable which GHC produces, 696,320 bytes, these extra bytes are insignificant.

Compilation overheads: there is no detectable difference in the size of the heap needed for compilation between the two profilers. The GHC default of 4 MBytes is used. The cost-centre profiler takes a total of 94.3 seconds to compile and link the program. This time is taken from an average of ten compilations using the Unix system time command. The cost-centre-stack profiler takes a total of 98.5 seconds to compile and link the code. This gives a time overhead of 5.57%.

Executable differences: the cost-centre profiler produces an executable file of 696,320 bytes. The cost-centre-stack profiler produces an executable file of 712,704 bytes. This gives a 2.35% size overhead when using the cost-centre-stack profiler. The execution time measured using the Unix time command is averaged over 10 executions of the program. The cost-centre profiler produces an executable which runs in 130.5 seconds; the cost-centre-stack profiler runs in 133.5 seconds. These run-time overheads are 2.29%[6]. The size of the heap needed to execute the program is the same for both profilers[7].

These results are encouraging for small programs, but the results of some larger programs must also be considered.

3.2 LOLITA results

The LOLITA system is one of the largest test cases available for profiling. The version of LOLITA which is profiled to gather these results contains 39,094 lines of Haskell code and 10,177 lines of C code. In addition, the system contains 6.79 Mbytes of data.

3.2.1 First example

The LOLITA system is interactive and offers a number of operations to its user. These operations will invoke different parts of the system and consequently will produce different results during profiling. Before any of these operations can be performed the system must load its semantic-net data. At the end of an execution, the LOLITA system saves the semantic-network data structure. As these two operations are required each time the LOLITA system is run, they provide the first test case for the cost-centre-stack profiler.

When the cost-centre-stack results are produced, the programmer's attention is drawn to the cost-centre stack with the highest costs:

```
<StaticNet_load_Ascii,StaticNet_sNetLoad,StaticNet_sInitData,
 Total_loadData,Okf_mapOKF,IMain_go,> with  157 TICKs
```

[6] The reader is referred to Sansom (1994) for the overheads in relation to un-profiled compilation and execution of programs.

[7] It is noted that any improvement made to the cost-centre profiler is likely to produce an overall improvement to the cost-centre-stack profiler.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

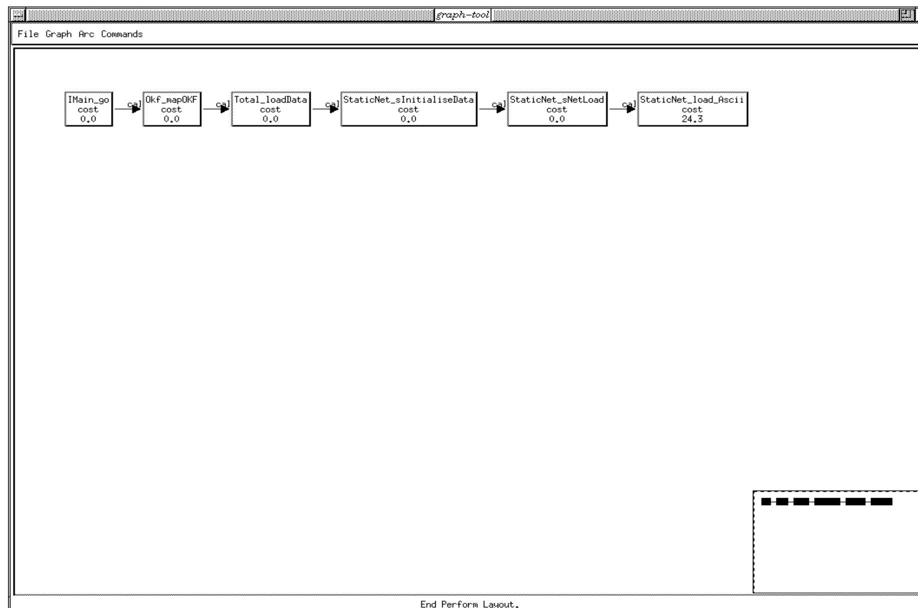


Fig. 11. Graph-tool view of the most expensive cost-centre stack.

At the head of this stack is the cost centre `StaticNet_load_Ascii`. This cost-centre stack alone accounts for 24.3% of the total execution time. These large costs are due to the loading of the LOLITA semantic-net data structure. The sequence of cost centres to this particular point in the program is clearly shown.

The investigation of the cost-centre-stack profile is facilitated by the use of the graph-tool and post-processor. There are a considerable number of cost-centre stacks to analyse and perhaps only the most diligent programmer is prepared to look through the cost-centre-stack output to find the expensive stacks.

The output is loaded into the graph tool; it contains 66 different nodes and 83 arcs between these nodes. For this reason the graph is large and cannot be seen in a single window display. The programmer is able to use the virtual display window in the bottom right-hand corner to view the remainder of the results.

The programmer may find that the inheritance and graph-tool functions are an easier way of managing the profiling results. Consider the following three examples of post-processing:

- Using the post-processing tool the programmer can select the cost-centre stack with the highest associated costs. This is effectively the most computationally expensive part of the code. This information is straightforward to interpret. Fig. 11 shows this function applied to the LOLITA results; there can be no question as to where in the program the largest amount of time was spent. Other post-processing functions could conceivably be developed which would add arms to the graph one by one, corresponding to the descending order of costs from the cost-centre-stack profile.
- A simplified view of the proposed function above is to display only the parts of the graph which have non-zero costs. A reduction of the graph of the

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

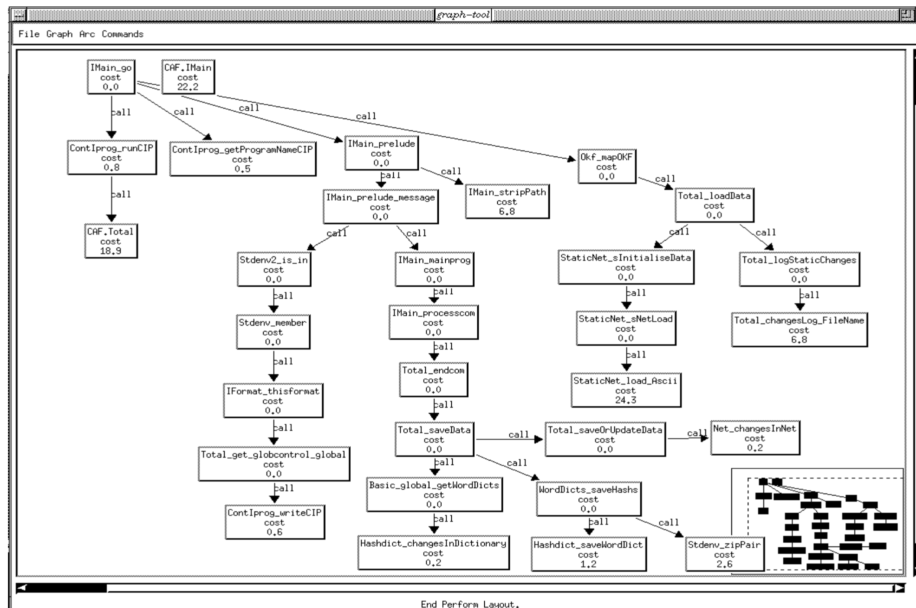


Fig. 12. Graph-tool view of all stacks with non-zero costs.

LOLITA system, to only those cost-centre stacks with associated costs, is shown in Fig. 12. The graph is reduced by more than a half; it now contains 31 cost centres and 30 arcs. It is clearly easier to read the results in this form as they appear on a single screen. In general, a programmer will only be interested in a profile of the program which shows the actual program costs.

- Post-processing also allows the programmer to select particular functions which he is interested in profiling. This facility is demonstrated on the LOLITA cost-centre-stack profile by selecting the following four cost centres: `IMain_go`, `Total_loadData`, `IMain_prelude` and `Total_saveData`. It may be useful for the programmer to gather profiling costs in terms of these four functions, as it allows the developers to see how much time is spent loading and saving the semantic net; how much time is spent in the prelude function of the `IMain` module and how much time is spent in the main function go. Those functions which are not selected have their costs subsumed by those functions which are selected; see Fig. 13. The costs do not account for 100% of the overall execution costs as some of the results are attributed to constant applicative forms (CAFs); these are top-level values which are not functions, `x=[1..]`, for example.

These post-processing facilities allow the programmer to explore the profiling costs after program execution. In this example, the programmer is able to see that loading the semantic-net data accounts for 31.1% of the total execution time. The LOLITA prelude accounts for 7.4% of the program costs; this involves formatting and printing the credit information and information regarding the authors of the system. At least 4.2% of the program's execution time is due to saving the semantic-

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

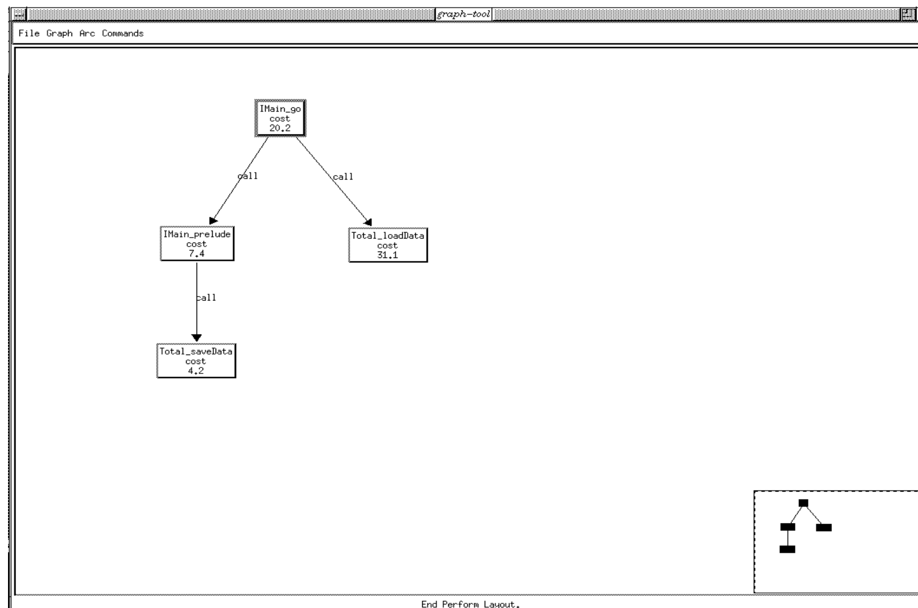


Fig. 13. Graph-tool view with selected cost centres.

net structure. Some of the saving costs (18.9%) have been attributed to a CAF in the `Total` module; this can be seen in Fig. 12, but not in Fig. 13. It is expected that later versions of the cost-centre-stack profiler will improve this situation; later versions of the GHC cost-centre profile are more explicit regarding the cost information for CAFs[8]. A further 20.2% of the program costs are subsumed to the cost centre `go`; this corresponds to the `main` function of LOLITA.

These results show a clear mechanism by which a programmer can collect and view profiling results. The post-processing facilities are clearly useful in such a large example and the way in which they are able to filter large collections of results gives the cost-centre-stack profiling scheme potential.

3.2.2 Second example

The second example considers a LOLITA function which makes use of a far greater percentage of the total code. Template analysis takes as its input a passage of text. This is then parsed and semantically analysed to produce a network of semantic nodes from which information can be scanned to match a collection of templates.

Two sets of input data were tested. The first was a passage of text taken from the *Daily Telegraph* newspaper concerning an IRA terrorist incident (*i*); it contained 74 words. The second piece of data was the sentence “The cat sat on the mat” (*ii*).

The first set of data clearly required more processing than the second, though using these two sets of data allows the profiling overheads required for each to be compared. Most of the time taken in (*ii*) is due to the loading of the semantic network.

[8] Now that the prototype has been tested and has provided satisfactory results, it is proposed that the cost-centre-stack profiler is implemented on GHC 2.02.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

Table 1. *LOLITA results for template analysis*

Input	Comp. time	Exec. size	Run-time	#Stacks	Depth	#Push
(*i*)	50123.7 (10.2%)	41279488 (7%)	110.9 (70.6%)	1807	112	278081
(*ii*)	same	same	55.4 (130.8%)	1766	94	12321

Table 1 shows: the input data used; the time taken to compile the LOLITA system using the cost-centre-stack profiler (the difference between this and the cost-centre profiler is shown in brackets); the size of the executable file produced by the cost-centre-stack profiler (the difference is again shown in brackets); the template analysis runtime (again the difference is in brackets); the number of cost-centre stacks produced as output; the depth of the largest cost-centre stack and the number of Push operations performed during program execution.

As expected the larger input causes more computation and as a consequence of this the mechanics of cost-centre-stack profiling are considerably more detailed - this is shown by the number of Push operations performed during each execution. The depth of the largest cost-centre stacks are similar for both sets of input; 112 and 94 cost centres respectively. Since large stacks are constructed for the small input as well as the large, many of the 265,760 Push operations (the difference between the two tests) will be performed as fast look-ups in cost-centre index tables. It is the speed and frequency of this look-up which is key to the success of the cost-centre-stack profiling technique. In effect, the Push operation builds a representation of the dynamic paths through the program in the cost-centre-stack table. The fact that a 20-fold increase in the number of Push operations between inputs (*ii*) and (*i*) leads to only a 2% increase in the number of stacks suggests that there are relatively few of these paths, and that much of the time spent in a program with larger input data is spent traversing paths which have already been encountered.

As well as the post-processing functions shown above, the cost-centre-stack profiler produces accurate inheritance results. These are particularly useful in the analysis of large graphs. There are two observations which can be made about inherited results:

- If a cost centre has a small number of inherited costs and is itself inexpensive then it is unlikely that any performance improvement made to this function will help to improve the program. This is not true of the cost-centre profiler where cost centres with low or even zero costs often contain a performance bug.
- Conversely, if a cost centre has a large inherited cost and is itself inexpensive then it may well be worth some attention, as this function may be the cause of large costs lower down in the program graph. This is different to the way in which the programmer would interpret a flat profile of the program.

Using these observations and the post-processing facilities, the programmer is able to perform a comprehensive analysis of his program. This will enable improvements

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

to be made to parts of the program without any further compilation, execution or profiling of the code.

3.3 `nofib` ***Results***

The previous results are further supported by the results of testing the cost-centre-stack profiler on the `nofib` benchmark (Partain, 1992). The `nofib` benchmark suite specifically consists of:

- Source code for real Haskell programs which can be compiled and run.
- Sample inputs (workloads) to feed to the compiled programs, along with the expected outputs.
- Specific rules for compiling and running the benchmark programs and reporting the results.
- Sample scripts showing how the results should be reported.

Those programs included in the `nofib` suite are divided into three subsets, Real, Imaginary and Spectral (between Real and Imaginary). The results displayed in this paper are of Real and Spectral programs; this means that they perform a useful task and are not implausibly small (or large). They are also written by someone trying to get a job done, not by someone trying to make a pedagogical or stylistic point. The results of Imaginary programs such as `queens` and `fib` are avoided.

The version of the `nofib` suite used in these tests dates from June 1996. All the tests were performed on the same machine[9]. The GHC optimiser (`-O`) was used during compilation and compile-time flags were set so that all top-level functions were profiled (`-prof -auto-all`). The programs were run with the time profiler (`-pT`) and the stats option (`-s`) so that the heap and time usage could be recorded.

In the majority of cases the supplied input was used during program execution, although some of the input data was extended to increase the runtime of the programs. The only other changes made to the programs were for debugging purposes (incorrect Makefiles, etc.). Not all of the programs included in the suite compiled correctly; some required a more up-to-date version of the compiler (0.24+) and some of the programs had files missing. All of the programs which compiled and ran correctly under GHC 0.22 are included, that is to say, this data was not selected on the basis that it produced favourable results.

For each program tested, the results of compiling and running the program under the cost-centre-stack compiler have been recorded. The difference in the overheads of the cost-centre-stack compiler and a standard version of GHC 0.22 (using the `-prof -auto-all` compiler flags) is shown in brackets. Statistics recorded include compile-time, runtime, the number of cost centres in the program and the number of Push operations performed by the cost-centre-stack profiler. The difference in the total heap usage is not shown as in each case these overheads were negligible. The results can be seen in Table 2. Analysis of the results shows the following:

[9] System Model : SPARCclassic, Main Memory : 96 MB, Virtual Memory : 353 MB, CPU Type : 50 MHz microSPARC, ROM Version : 2.12, OS Version : SunOS 4.1.3C.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

Table 2. `nofib` *benchmark results*

Program	Comp.-time sec. (diff.)	Run-time sec. (diff.)	Cost centres	#Push
Real subset				
`ebnf2ps`	2518.5 (13.5%)	3.9 (84.8%)	225	34335
`gamteb`	816.0 (13.2%)	216.3 (100.1%)	57	356142
`gg`	1097.7 (11.1%)	14.4 (73.2%)	133	76272
`maillist`	106.8 (3.8%)	20.1 (6.0%)	10	5367
`mkhprog`	476.3 (13.9%)	0.4 (42.8%)	30	134
`parser`	1331.9 (7.7%)	79.5 (384.7%)	78	1375957
`pic`	475.3 (6.5%)	11.9 (6.25%)	26	6885
`prolog`	442.0 (12.6%)	4.3 (358.3%)	64	40847
`reptile`	1116.2 (9.9%)	7.3 (38.0%)	253	30051
Spectral subset				
`ansi`	141.2 (5.7%)	0.7 (133.3%)	26	113
`banner`	265.6 (0.2%)	0.6 (10.9%)	9	3359
`clausify`	132.7 (16.6%)	14.8 (33.3%)	26	693542
`eliza`	284.2 (7.12%)	2.6 (144.8%)	17	24696
`minimax`	379.0 (6.6%)	5.0 (152.3%)	40	114384
`primetest`	216.6 (6.0%)	153.3 (2.4%)	21	24062

Compile-time: between 3.8% and 13.9% overhead. This is due to the time needed to produce the larger executable file. The size of the executable files was expected to be slightly larger because of the changes made to the compiler optimiser and to the run-time system.

Heap usage: no detectable difference, as expected, since most of the changes made to the compiler are to the run-time system.

Run-time difference: this is where the most overheads are anticipated, as most of the profiler changes are to the run-time system. These range from 2.4% to 384.7%. These overheads are dependent on the structure of the program (see section 4). Even when the run-time overheads are 384% (`parser`), this only means an extra 59 seconds of execution time, which accounts for just 4.4% of a single compilation of that program.

These results show that the cost-centre-stack profiler should be used if the cost centres are going to be moved one or more times in the analysis of a program. If this is the case, a substantial amount of time will be saved. The cost-centre-stack profiler also allows a post-mortem manipulation of the profiling results. Information can be selected on different parts of the program without the programmer needing to physically alter the original program. This is of considerable added benefit to the programmer as any modification to the source code for profiling is likely to introduce errors.

The relation between, the number of cost centres and Push operations, and the overheads is discussed in the next section.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

4 Complexity analysis

How can the difference in run-time overheads found in the testing of the cost-centre-stack profiler be explained? The answer is that the run-time overheads are dependent on the structure and style of the program. For example,

- the complexity of the cost-centre-stack table will increase with the number of arcs in the call-graph. The greater the functional dependency, the greater the overheads involved in creating the cost-centre stacks;
- the more cost centres there are per unit of code, the greater the overhead of managing the cost-centre stacks. The number of top-level functions may be increased by the programmer's style, for example, if the programmer does not use many local function definitions.

It is important to note that it is not simply the size of a program which increases the overheads. This is shown by the LOLITA results, for example, where the overheads are lower than the overheads of programs hundreds of times smaller. It is the possibilities allowed in the call-graph, which are fulfilled at runtime, that increase the overheads.

Of course, this analysis (as in the `nofib` results) is based on the assumption that all top-level functions are profiled (`-auto-all`). This need not always be the case. It is quite possible to profile explicitly-annotated cost centres in addition to the top-level functions. It is also possible to profile only functions exported from a module (`-auto`) or only explicitly-annotated cost centres. Since it is possible to de-select cost centres during the post-processing phase, the only reason for reducing the number of cost centres at compile-time is to decrease the profiling overheads. Fewer cost centres will mean fewer push operations and fewer and smaller stacks. A case in which this has been necessary has yet to be found.

A worst- and average-case analysis of the cost-centre-stack profiler may be found in Jarvis (1996) – the analysis also considers the overheads of this profiling scheme according to different program structures. For example if the call-graph is a tree, where each function is called by only one other function, then the complexity of the algorithm is logarithmic over the number of cost centres in the program.

5 Debugging and tracing

There are a number of further applications of this cost-centre-stack technique, two of which are the debugging and tracing of Haskell programs.

There are two types of error found in Haskell programs which are particularly awkward for the programmer to detect:

- The first is non-termination as there is usually no helpful information given as to why this is the case.
- The second is the familiar `head []` error. When this error is presented it is often very difficult to determine which the caller was. Part of the problem is that the programmer wants to know what built the thunk (`head []`) rather than the function which demanded it.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

It is proposed that the cost-centre-stack profiler be modified so that it outputs the current stack of cost centres when an error occurs in a program or the user interrupts its execution. This means that the location of an error can be determined immediately; it is also possible to trace the path which the program took to that erroneous piece of code. This limits the search required by the programmer and in initial tests has provided invaluable information in the debugging of the LOLITA system.

It is also possible to output the cost-centre stacks as the program is executing. Although this provides a lot of information it does allow the programmer to watch the order of evaluation of expressions in the execution of a program. Alternatively the programmer could stop the program, pressing control-C, and the current cost-centre stack could be printed. In the past, the only way to see the sequence of lazy evaluation was to watch the program stack. This was not easy and was difficult to interpret even for the experienced programmer.

The cost-centre-stack approach to debugging and tracing is very flexible as the programmer can insert cost centres into the code where he thinks it is necessary, therefore controlling the output of the debugger just as a C programmer might add `printf` statements to his code. Integrating the tracing and debugging functions with the post-processor may provide the programmer with a useful environment in which to profile, view and debug his Haskell programs.

6 Conclusions

The development of the cost-centre-stack profiler was based on the results of a series of case studies implemented over a three-year period. The case studies investigated the profiling of the LOLITA system, a large-scale lazy functional system written in 47,000 lines of Haskell code. This study highlighted a number of problems with the current profiling tools and in response to these the cost-centre-stack profiler was designed.

The cost-centre-stack profiler collects results which can then be post-processed after the execution of a program. The post-processor implements a scheme whereby the programmer can select and reselect cost centres in his code and view the results accordingly. This enables the results to be displayed at different levels in the program without any further compilation or execution of the code.

The implementation of the cost-centre-stack profiler and the post-processor provides a number of benefits to the programmer:

- The new method of profiling provides an opportunity for a reduction in the time needed to profile Haskell programs.
- The new method of profiling extends the results presented by previous profilers in so much as the accurate inheritance of shared program costs can be achieved without having to recompile and rerun the program.
- The new method of profiling provides these new facilities without imposing an unacceptable overhead on the compilation or execution of a Haskell program.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press

It is also considered that this method will offer assistance in the debugging and tracing of Haskell programs.

References

Clack, C., Clayman, S. and Parrott, D. J. (1995) Lexical Profiling: Theory and practice. *J. Functional Programming*, **5**(2).

Long, D. and Garigliano, G. (1994) *Reasoning by Analogy and Causality: A Model and Application*. Ellis Horwood.

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

Jarvis, S. A. (1996) Profiling Large-scale Lazy Functional Programs. *PhD Thesis*, University of Durham.

Jarvis, S. A. and Morgan, R. G. (1996) The results of: Profiling large-scale lazy functional programs. *Implementation of Functional Languages: Lecture Notes in Computer Science 1268*, pp. 200–221. Springer-Verlag.

Launchbury, J. (1993) A natural semantics for lazy evaluation, *Proceedings of 20th ACM Symposium on Principles of Programming Languages*, Charlotte. ACM.

Morgan, R. G. and Jarvis, S. A. (1995) Profiling large-scale lazy functional programs. In A. P. W. Bohm and J. T. Feo (eds.), *Proceedings of High Performance Functional Computing*, Lawrence Livermore National Laboratory, USA, pp. 222–234.

Partain, W. (1992) The `nofib` Benchmark Suite of Haskell Programs. Department of Computer Science, University of Glasgow.

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

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

Sansom, P. M. and Peyton Jones, S. L. (1995) Time and space profiling for non-strict, higher-order functional languages. *22nd ACM Symposium on Principles of Programming Languages*, San Francisco, CA.

https://doi.org/10.1017/S0956796898003013 Published online by Cambridge University Press