UNIVERSITY
of
GLASGOW

Computing Science

Ph.D. Thesis

Execution Profiling for Non-strict Functional Languages

Patrick M. Sansom

Submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Execution Profiling for Non-strict Functional Languages

by

Patrick M. Sansom

Submitted to the Department of Computing Science
on 29th April, 1994, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

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

This thesis examines the difficulties of profiling lazy higher-order functional languages and develops a profiling tool which overcomes them. It relates information about both the *time* and *space* requirements of the program back to the original source expressions identified by the programmer. Considerable attention is paid to the cost semantics with two abstract cost semantics, *lexical* scoping and *evaluation* scoping, being investigated. Experience gained from the two profiling schemes led to the development of a hybrid cost semantics. All three schemes are described and compared in a single formal framework.

These abstract cost semantics are mapped onto an operational semantics and an implementation based on the STG-machine is developed. The manipulation of cost centres is made precise by extending the state-transition operational semantics of the STG-machine.

The profiling tool has been incorporated into the Glasgow Haskell compiler **ghc**. Our approach preserves the correct cost attribution of costs while allowing program optimisation to proceed largely unhindered. So far as we know **ghc** is the only lazy functional language compiler to support source-level *time* profiling. The use of the profiler has lead to significant performance improvements in the compiler itself and other large application programs.

Thesis Supervisor: Prof. Simon L. Peyton Jones
Title: Head of Planning Unit, Department of Computing Science

Acknowledgements

I would like to thank my supervisor, Professor Simon Peyton Jones, for his guidance and support during this research. I would also like to thank my colleagues in the Functional Programming Group for listening to my ideas and providing invaluable feedback.

This thesis is a direct result of my involvement in the GRASP project and in particular, my participation in the design and implementation of the Glasgow Haskell compiler. I'd like to thank all the members of the GRASP team for providing the framework that made my research possible. Special thanks to Will Partain for answering an endless stream of questions and slaving away on my behalf.

This research would not have been possible without financial support. I am very grateful to the Commonwealth Scholarship Commission for the scholarship that made all this possible. I am also grateful to John Launchbury for giving me time off to complete the writing up of this thesis.

Finally, I'd like to thank Lisa Schroder for all her support.

To the fond memory of my beloved grandfather

C.E. Carter

Contents

List of Figures

Chapter 1

Introduction

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

1. Identify the execution bottlenecks or "critical parts" of the program that account for much of the time and space used. This allows effort spent improving the program to be focussed on parts of the program where it will be of greatest benefit.

2. Identify any inefficiencies present in the bottlenecks thus identified. These may range from "hidden" *space leaks* (Peyton Jones [1987]), caused by the subtleties of the method of evaluation, to inappropriate choices of algorithms and data structures.

Once this is done, alternative, more efficient, solutions can be proposed and evaluated.

Conventional languages provide profiling tools such as *gprof* (Graham, Kessler & McKusick [1983]) and *mprof* (Zorn & Halfinger [1988]) which attribute time usage and space allocation to the source code. This enables the programmer to identify the "critical parts" of the program being developed. However current functional language programming environments, especially for non-strict (so-called lazy) languages, lack equivalent tools.

As the use of lazy functional languages for applications programming has grown there has been a strong call for source-level profiling tools to aid the applications programmer in the identification of execution hot-spots and inefficiencies. The lack of these tools has severely hindered the use of lazy functional languages for real applications programming. This thesis attempts to address this shortcoming by developing suitable profiling tools for

the lazy functional language Haskell.

Though these tools include the identification of *time* spent in the different "parts" of the program, we are also interested in tools that identify the *space* usage. Unlike most conventional languages, functional languages provide an abstraction which hides the allocation and reclamation of data structures. This abstraction can result in unexpected spatial behaviour ranging from over-liberal allocation to so-called *space leaks*. Results from Runciman and Wakeling who have developed a heap profiling tool have indicated how revealing such information can be (Runciman & Wakeling [1993]; Runciman & Wakeling [1992]).

1.1 Scope

This thesis is concerned with the development of source-related performance profiling tools for sequential evaluation of lazy functional languages which are independent of a particular implementation (though the costs themselves depend on the efficiency of the implementation). It does not address the provision of performance statistics about a particular implementation nor the profiling of parallel evaluation.

Profiling is concerned with the detection of "performance bugs". We do not address the provision of more specific debugging tools for lazy functional languages. However, there is a close relationship between profiling tools and debugging tools: debugging tools may be required to identify the cause of a particular "performance bug" identified by a profiler; and profiling data may reveal algorithmic bugs in the program.

The emphasis of this thesis is on the underlying cost semantics and efficient implementation of the basic technology required to gather the profiling data. Though the presentation of this data to the programmer is also important we do not focus on this issue. Our implementation uses straightforward presentation techniques, making use of existing presentation tools where appropriate.

1.2 Main Contributions

This thesis develops a time and space profiling system for a compiled implementation of Haskell (Hudak et al. [1992]): a non-strict, higher-order, purely functional language. The

main contributions made are:

- The approach taken explicitly addresses the key problem of the identification of the different program "parts" for which statistics are accumulated. To this end the notion of a "cost centre" is introduced. Cost centres are used to identify the source code expressions of interest. They are introduced by annotating the source or automatically by the compiler. A simple, but effective, inheritance scheme is employed that attributes the costs of unprofiled functions to the cost centres where they are used.

- Considerable attention is paid to the meaning of the "cost of evaluating an expression". We develop a formal semantic model that makes precise which costs are attributed to which cost centre. This model provides a setting in which we explore, in a precise way, several different cost-attribution schemes.

- The abstract cost semantics are then mapped onto a push-enter semantic model which incorporates an argument stack. From this, STG-machine implementations for each of the profiling schemes are developed (Peyton Jones [1992]). These are formally presented as extensions to the state transition system for the STG-machine.

- An implementation scheme is presented that allows the *time* spent in a particular "part" of the program to be measured, even though lazy evaluation causes this execution to be interleaved with different parts of the program.

- The practicality of our approach is demonstrated by our implementation in the context of the Glasgow Haskell compiler — a state-of-the-art optimising compiler (Peyton Jones et al. [1993]). Our approach preserves the correct cost attribution of costs while allowing program optimisation to proceed largely unhindered. Programs compiled with profiling enabled report both time and space usage to the programmer. The basic execution overhead is about 65%.

- The practical use of the profiling tool is demonstrated with results from the profiling and improving of the compiler itself.

One of the major strengths of our profiler is that it has been incorporated into a widely-used, production-strength, optimising compiler. This provides programmers with a profiler

that they will actually use. It makes it more than an interesting, but obscure, research toy. It also highlighted the shortcomings in our early cost-attribution models and focussed attention on "what really matters".

1.3 Outline

This thesis begins with a general discussion of execution profiling, describing the task of execution profiling, highlighting the requirements of a profiling tool, and briefly examines the different profiling tools that exist for conventional languages (Chapter 2). We then go on to discuss the particular problems that have to be overcome when profiling lazy, higher-order functional languages and describe the profiling tools that currently exist for these languages (Chapter 3).

In Chapter 4 we introduce "cost centres". After discussing the principles of cost attribution we extend an abstract semantics with a well defined notion of cost attribution. Two different cost semantics are developed and compared. This leads to the development of a third, hybrid cost semantics which incorporates the desirable properties of both semantics.

Chapter 5 describes the implementation of our profiler. This has three main components:

- The appropriate cost attribution is preserved across the transformation phases of the compiler.
- The development of profiled STG-machine implementations for each of the three cost semantics. On the way to our STG-machine implementations we develop abstract push-enter semantics with equivalent cost attributions. The gory details of the STG-machine are relegated to Appendix A.
- Modifications to the runtime system to record the necessary attribution information and gather the required profiling data.

The output of profiler is described in Chapter 6. This includes both time and space profile. Example profiles are generated using the optimised `clausify` program (Runciman & Wakeling [1993]).

The practical use of the profiler with large applications is demonstrated in Chapter 7. We present detailed results obtained while profiling the compiler itself and report on the

experiences some other users have had using the profiler.

Finally, in Chapter 8, we present our conclusions.

Chapter 2

Execution Profiling

> *Programming differs from most other crafts in one notable way: it is purely descriptive. The job has been done when the way to do it has been described. Compared to building and tuning an engine, programming is a very non-physical experience. Inefficiencies are not betrayed by great vibrations of the computer — in fact we seldom even see the computer. What is missing in this craft is feedback.*
>
> D Ingalls [1972]

An *execution profile* attempts to provide this feedback by reporting to the programmer information that highlights any inefficiencies within their program. This enables the programmer to direct any effort spent improving the program to the parts where it will be of most benefit.

The potential benefits of execution profiling were first highlighted by Knuth [1971]. He reported the results of a study of the behaviour of FORTRAN programs drawn from a number of applications, observing that "less than 4 per cent of a program generally accounts for more than half of its running time." The identification and improvement of this 4 per cent can have a dramatic effect on overall performance. Knuth reported that the use of a profiler "made it possible to double the speed of FORDAP [the profiling pre-processor] in less than an hours work." Indeed Knuth's profiling experiences led him to conclude that "profiles should be made available routinely to all programmers by all of the principal software systems."

The benefits of using a profiling tool have been repeatedly highlighted with large

speedups being reported with relatively little effort when that effort has been directed at *known* trouble spots revealed by a profiler (Darden & Heller [1970]; Ingalls [1972]; Knuth [1971]; Ripley & Griswold [1975]; Satterthwaite [1972]; Sites [1978]; Waite [1973]). Bentley [1987] gives a delightful account of some profiling "pearls".

Before going on to describe conventional profiling systems I first pause to consider the way programmers use profilers, and the requirements of any profiling system.

2.1 Developing Efficient Programs

The availability of profiling tools offers a new approach to the development of efficient programs. Instead of writing code obscured by the concern for efficiency, the programmer can initially write simple, maintainable code without much concern for efficiency. Once completed and debugged the performance of the program can be profiled, and effort spent improving the program where it is deemed necessary.

The use of this approach to program development was first advocated in the early seventies. Darden & Heller [1970] used it in the development of their Algol compiler. Ingalls [1972] introduces the idea in a more general discussion of the benefits of profiling. It also forms the basis of Bentley's book "Writing Efficient Programs" (Bentley [1982]).

This approach to development has a number of advantages:

- Added complexity caused by, possibly unnecessary, efficiency concerns during the initial development is avoided. This significantly reduces the initial development costs.

- Program efficiency is only considered where it is important. The time to rewrite the important sections is low as there is very little code that needs this attention.

- Maintenance costs are reduced, since most of the code remains clean and easy to understand. Where efficiency modifications have been made the original code can often form the basis of the documentation for the more obscure, but efficient, code.

2.1.1 Profiling programs

Figure 2.1 depicts the profiling cycle. The most important task in improving the performance of a program efficiently is the identification of the "bottlenecks" or "hot spots"

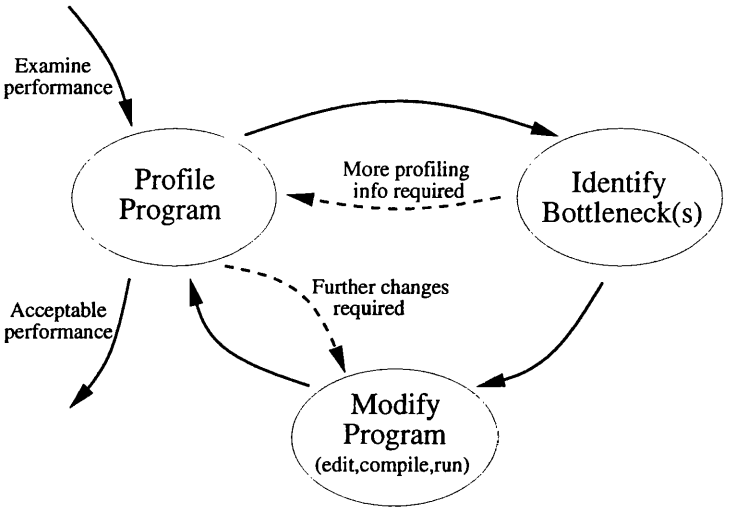


Figure 2.1: Improving Performance — The Profiling Cycle

in the program. A lot of effort can be wasted improving parts of a program that the programmer "thought" were sources of inefficiency but actually consumed a relatively small amount of resources. The main function of a profiling system is to provide the programmer with the information that enables the identification of any bottlenecks. Additional information may still be required to identify the cause of a particular bottleneck. This may range from a more detailed program profile to specific program generated trace data.

Once the causes of the hot spots in a program have been identified the offending algorithms, data structures, and/or code can be improved. The effect of the modifications should then be measured to determine the effect the changes had on the performance. In particular the programmer must decide if a particular bottleneck has been removed or if further improvements are still required.

As the performance is improved other bottlenecks may be identified and subsequently improved. This process can continue until the programmer is satisfied with the performance, or deems the expected effort required to undertake any further improvements to outweigh the expected performance benefits.

2.2 Requirements of Profiling

Profiling is essential if a program's performance is to be improved efficiently. The information gathered about the execution must provide the basis for answering the key questions:

- Where are the execution bottlenecks?
- What was the effect of a particular modification?

For a profiler to be of use it must accurately report data about the execution of the program presented in a form that enables these questions to be easily answered. In particular:

- The profiler must measure the distribution of the key program resources.
- The measurement data must be related to the program source in a way that is meaningful to the programmer.

In addition, the profiling system must also provide the facilities to aid the programmer in the identification of the causes of a particular bottleneck.

2.2.1 What should be measured

If a profile is to identify program bottlenecks then it must measure the use of resources that are likely bottlenecks, and attribute the demand for the resource to the appropriate program part. The most obvious resource is execution time. A number of techniques have been used to measure execution time. These are described in Section 2.3.

However, execution time is not the only possible bottleneck. Memory is also a limited resource. Excessive dynamic memory requirements result in an increased amount of time spent in the garbage collector. In a virtual memory system, these memory requirements may lead to thrashing. If the memory requirements are degrading performance they must be addressed directly. An understanding of a program's memory requirements requires a very different profile: see Section 2.4.

2.2.2 How should data be presented

Any execution data must be related to the source code responsible for the costs. This is absolutely essential since it is the source code that the programmer has to modify. The data must be presented in a way that draws the attention of the programmer to the performance problems. Typical presentations included:

- A profile summary reporting each source-level function, possibly ordered by the execution cost conveyed by the profiling data.

- Annotated source listings. These can be easily scanned for smaller programs, but for a larger program are only useful once the costly procedures have been identified.

In addition, as a program modification often affects an entire logical component, it is desirable to be able to instruct the profiler to aggregate the measurement data into logical groupings that reflect the program structure. Example logical groupings include:

- The total cost of a function including all sub-function calls.
- The cost of *all* the operations provided by an abstract data type.

Such a facility enables the total cost of a logical component to be easily determined and compared with another implementation of the same component (see Section 2.5).

2.2.3 Understanding the bottlenecks

Once a bottleneck has been identified the programmer still has to find the cause of the bottleneck before a solution can be developed.

The programmer may gain additional understanding about the execution of their program from a whole range of execution data. Examples include: execution counts of functions or source lines; I/O activity; and memory allocation. Though all this information may be of interest a profiling system must be careful not to swamp the programmer with unnecessary data, providing the additional data only if requested.

Additional tools that aid this task are more in the realms of debugging and execution tracing than profiling, but are still an essential part of a programmer's profiling armoury. In fact, the process of improving performance could be termed "performance debugging".

2.2.4 Constraints on profiling

Though the act of profiling changes the execution behaviour of a program, the execution data reported should (as far as is possible) accurately reflect the execution that would occur during normal execution. In particular:

- Compiler optimisations must not be turned off. If normal execution is optimised then the profiled execution should also be optimised. Unfortunately this makes the job of relating the execution data back to the original source much more difficult as the source may no longer reflect the execution that actually occurs. This is a particularly

awkward problem for very high-level languages (Appel, Duba & MacQueen [1988]). Solutions require the integration of profiler and compiler.

- Overheads introduced by the profiler should, where possible, be discounted in any measurement data reported.

- If the profiler includes real execution timings, execution overheads must be minimised to avoid distorting these timings. However, since we are interested in the relative timings, a constant factor overhead across *all* execution is acceptable as long as any variation in the overhead is small.

In addition, the profiling overheads must be small enough to permit the profiling of expensive programs running on real data sets — it is exactly these programs that need to be examined and improved! Typical overheads for conventional profilers are between 5% and 100% with anything less than 30% generally considered quite acceptable.

2.3 Time Profiling Systems

Profiling systems have been developed for many different languages and execution platforms. The vast majority, especially for conventional programming languages, have been concerned with the profiling of execution time. Though the number of execution time profiling systems developed is quite large, the profiles produced and techniques used fall into three basic categories.

2.3.1 Frequency counts

Frequency count profiling inserts counters in each basic block of the program in order to determine the number of times each statement is actually executed (Coutant, Griswold & Hanson [1983]; Foxley & Morgan [1978]; Ingalls [1972]; Knuth [1971]; Lyon & Stillman [1975]; Satterthwaite [1972]; Wichmann [1973]). This provides a great deal of information about the execution of a program revealing the dynamic behaviour of the code being executed. As well as highlighting the inner loops, it reveals unexecuted code, and the dynamic behaviour of the algorithms used.

This can be augmented with an estimation of the execution cost of each statement to provide a cost oriented profile, identifying the expensive parts of the program (Ingalls

[1972]; Knuth [1971]).

2.3.2 Execution sampling

Execution sampling interrupts the execution of the program at periodic intervals recording which part or procedure of the program is currently executing (Appel, Duba & MacQueen [1988]; Brailsford et al. [1977]; Graham, Kessler & McKusick [1983]; Ingalls [1972]; Jasik [1972]; Knuth [1971]; Ripley & Griswold [1975]; UNIX Programmer's Manual [1979]; Waite [1973]). If the execution time is long enough to provide a significant number of samples, the data gathered gives a good indication of the relative execution times of the different parts of the program. Due to the random nature of the of the sampling process, two sampled profiles will not give identical results. This profiling scheme tends to be less precise but more realistic as it includes time that is spent in system (as opposed to user) subroutines.

2.3.3 Procedure timings

Procedure timing profilers insert statements that read a system clock at the entry and exit points of each procedure or program unit (Bergeron & Bulterman [1975]; Matwin & Missala [1976]; Wichmann [1973]). This enables the time spent in each procedure, either including or excluding any sub-procedure calls, to be determined. Unfortunately the cost of accessing the system clock is often prohibitively expensive and the accuracy of the profile is dependent on the resolution of the system clock.

2.4 Allocation and Memory Profiles

Most programming environments provide an automatic or explicit storage management system. Understanding the dynamic space or *heap* requirements of a program can reveal additional bottlenecks such as:

- Allocation hot spots.
- Large space requirements caused by the construction and retention of large, space-hungry data structures.

- Unidentified "memory leaks" caused by the failure to reclaim storage that is no longer required. In long-running programs such a "memory leak" can have a very serious effect on performance.

Though a time profile may reveal an excessive amount of time spent in the storage management routines, it does nothing to identify the source of a program's memory requirements. Similarly, statistics provided by storage management systems about the heap objects allocated and amount of live heap data processed at each garbage collection, also do nothing to identify the source of the memory requirements.

2.4.1 Allocation profiles

The simplest dynamic memory profile is an allocation profile (Coutant, Griswold & Hanson [1983]; Ripley, Griswold & Hanson [1978]; Zorn & Halfinger [1988]). This reports information about the allocation of dynamic memory, attributing it to the source location responsible for its allocation. A more detailed breakdown of allocation is provided by **mprof** (Zorn & Halfinger [1988]) which attributes allocation to the dynamic call sequence responsible.

Though an allocation profile may reveal the allocation hot spots, these do not necessarily correspond to the source of the long-lived dynamic data that happens to be consuming all the memory. It may be that the long-lived data is allocated by seemingly insignificant allocation site(s) that are not highlighted by the allocation profile. Identifying the source responsible for allocating the long-lived data requires yet another profile.

2.4.2 Leak profiles

Specialised profiles for identifying "memory leaks" in C programs, which use explicit dynamic storage management, are described by Barach & Taenzer [1982], and Zorn & Halfinger [1988]. They identify heap objects that are never deallocated and report the call sequence responsible for allocating them. Indeed, the problems associated with "memory leaks" in explicit storage management systems have resulted in a number of implementations incorporating (conservative) garbage collection schemes to remove the need for explicit deallocation (Bartlett [1988]; Boehm & Wuiser [1988]; Caplinger [1988]; Wentworth [1990]; Zorn [1992]).

2.4.3 Heap profiles

A more general tool, used in systems with implicit storage management, is the *heap profile*. This reports the *live* data occupying the heap, attributing it to the source location responsible for its allocation. Since the contents of the heap change over time the profile must describe the behaviour of the heap objects over time. This can be done by reporting summary statistics about the lifetime of the heap objects (Ripley, Griswold & Hanson [1978]) or by presenting a number of "snap shots" of the objects occupying the heap during execution (Ripley, Griswold & Hanson [1978]; Runciman & Wakeling [1993]).

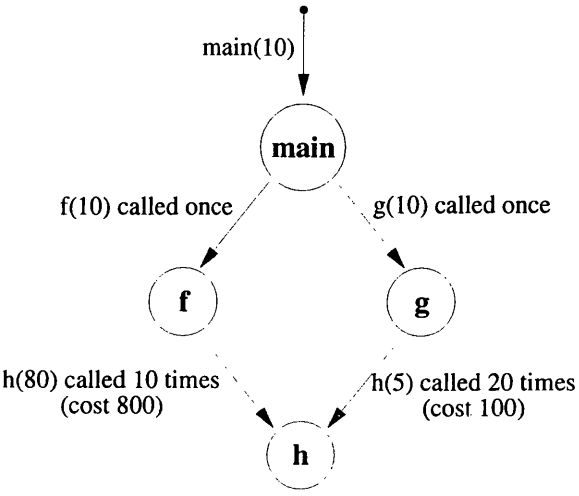
Due to the large amount of data such a profile can generate careful attention needs to be paid to the presentation of the profile and the facilities provided to select relevant data. These issues are addressed by the **hbc/lml** heap profiler (Runciman & Wakeling [1993]) (see also Section 3.3).

2.5 Aggregation and Inheritance

Section 2.2.2 described the need to provide a profiling scheme that aggregates the cost of the logical components of a program. This is particularly important when profiling large programs, since reporting isolated data for each function[1] is cumbersome and unilluminating.

Figure 2.3 shows a very basic flat profile for the call graph presented in Figure 2.2. It reveals that 900 time units are spent in **h**. If we wish to improve the performance we could optimise **h** directly and/or improve the way in which **h** is called. Unfortunately this flat profile does not show which functions were responsible for calling **h**, nor the costs associated with the various call sites. What we would like to be able to determine is the total costs of **f** and **g**, including the costs incurred executing **h**. This would reveal that the execution of **f** is the bottleneck, costing a total of 810 time units. This requires the costs of any sub-functions called to be attributed to the calling function as well.

[1] We use the term *function* to refer to the basic program unit — normally a *function* and/or *procedure* depending on the programming language being profiled.



The argument passed to each function represents the basic cost of execution, excluding any sub-function calls. Observe that **h** is called from both **f** and **g** with substantially different execution costs.

Figure 2.2: Example Call Graph

Function	#calls	Time
main	1	10
f	1	10
g	1	10
h	40	900

Figure 2.3: Flat Time Profile

		Statistical Inheritance	Accurate Inheritance
Parents Function Children	Called/Total Total Called/Total	Inh Time Inh	Inh Time Inh
main	1	930	930
f	1/1	310	810
g	1/1	610	110
main	1/1	310	810
f	1	310	810
h	10/30	300	800
main	1/1	610	110
g	1	610	110
h	20/30	600	100
f	10/30	300	800
g	20/30	600	100
h	30	900	900

Figure 2.4: Call Graph Profiles — Statistical and Accurate Inheritance

2.5.1 Call graph profiling

A much more detailed profile can be generated if information about the arcs of the call graph, rather than just the nodes, is gathered. Instead of recording the number of times each function is called a call count is associated with each call site. This enables the functions responsible for calling each function to be identified.

Statistical inheritance

The call graph information can also be used to propagate an approximate cost up the call graph by apportioning the time spent in a particular function to its various callers. This *statistical* inheritance scheme is used by **gprof** (Graham, Kessler & McKusick [1983]). The example inheritance profiles in Figure 2.4 show the total cost for each function. The costs inherited from each child are displayed below the entry for the function, and the costs inherited by each parent are displayed above the entry.

Unfortunately the accuracy of this scheme relies on the assumption that the average cost of a call to a function is independent of the call site. If this is not the case incorrect

costs are attributed to the calling function.[2] For example, the statistical inheritance column in Figure 2.4 reports the total cost of **g** as 610 units where as its actual cost is only 110 units. This error arises because the cost of calls to **h** from **g** are considerably less than the mean cost of all calls to **g**.

Accurate inheritance

Accurate inheritance of costs is possible if the costs are attributed directly to *all* the caller/callee pairs on the current call stack during execution. This scheme is used by **mprof** (Zorn & Halfinger [1988]) which attributes memory allocation data to all the caller/callee pairs on the call stack (up to a maximum depth of 5). Unfortunately the overheads involved in such a scheme, especially if data points are frequent, are very large.

Coping with cycles

Any call graph inheritance scheme has to cope with cycles in the call graph arising from recursion in the executing program. Costs should not be attributed multiple times to the same function. Both **gprof** and **mprof** solve this problem by collapsing recursive cycles into a single node in the graph. This can result in a loss of information, but seems to be an adequate solution.

2.5.2 Subsuming costs

Limited, but accurate, inheritance can be achieved without large overheads if the costs of any unprofiled functions are attributed directly to the calling function i.e. the costs of any unprofiled sub-functions are *subsumed* by the caller as if the sub-function was part of the caller. During execution profiling data is attributed to the profiled function that is deemed to be currently executing (though an unprofiled sub-function may actually be executing). This scheme was first used in the New Jersey SML profiler (Appel, Duba & MacQueen [1988]) which introduced the notion of the *current (profiled) function.*

[2] Statistical profilers like **gprof** usually collect accurate timing information for each parent-child call count. This provides accurate inheritance to the immediate parent, but the inheritance approximations still arise when costs are inherited to grandparents.

Function	#calls	Time
main	1	10
f	1	810
g	1	110
h	—	—

Figure 2.5: Subsumed Profile

Unfortunately this scheme results in the loss of information about the unprofiled sub-functions. At least the loss of information is controlled by the programmer who identifies the unprofiled functions. The up-side is that it provides more accurate information about the total costs of the calling functions. If all the sub-functions called are unprofiled the total cost reported is indeed accurate. For example, marking **h** as an unprofiled function would result in the basic profile presented in Figure 2.5. Though we have lost information about **h** we see accurate costs reported for **f** and **g**. Of course, we can still generate information for **h** (as in Figure 2.3) by running the profiler again with **h** marked for profiling. These two profiles can then be compared to determine the distribution of the costs of **h** between the different call sites. (Though this does not provide an accurate breakdown of the call-site counts.)

In addition, this profiling scheme can be combined with information about the call graph of profiled functions. In particular, statistical inheritance can still be used to propagate the costs of the profiled functions, providing approximate total costs for functions that still call profiled sub-functions.

2.5.3 Module structure

A program's module structure may also be used to provide an alternative grouping of costs. The module structure usually reflects the logical structure of various program components. Summing the cost attributed to all the functions provided by a module is an easy way of reporting the total cost of a logical component such as an abstract data type. This task has usually been left to the programmer using the profiler, though the profiler may aid this process by sorting the profiling data by module.

2.6 Profilers Today

The conventional profiles around today have not moved much beyond the technology developed in the seventies. The basic source related feedback about the execution behaviour of a program, such as statement counts and/or procedure costs, possibly with inheritance, is as invaluable to the programmer as ever. The only advances seem to have been in the provision of integrated programming environments making the profiling information more readily available.

More recent profiling research has moved towards interactive visualisation or monitoring of program behaviour (see, for example, Jeffrey [1993]). However, this is beyond the scope of this thesis.

Chapter 3

Lazy Profiling

> *In considering the total cost of computing, people began to observe that program development and maintenance costs often overshadow the actual costs of running the programs. Therefore most of the emphasis in software development has been in making programs easier to write, easier to understand and easier to change. There is no doubt that this emphasis has reduced total systems cost in many installations, but there is little doubt that the corresponding lack of emphasis on efficient code has resulted in systems which can be greatly improved, and it seems to be time to right the balance.*
>
> DE Knuth [1971]

Lazy functional programming environments have typically provided few profiling tools despite the fact that they are more prone to unexpected "performance bugs" than their imperative counterparts. This is largely because the task of producing a useful profile for lazy functional programs is more difficult than doing so for a program written in a more conventional, strict language.

3.1 Performance of Lazy Functional Programs

Functional languages provide the programmer with a high level of abstraction from the computer architecture on which the program is run. They enable the programmer to concentrate on expressing the solution to the problem in a declarative manner, without worrying about low-level execution details. The abstraction provided can significantly reduce the costs of developing large applications. Page & Moe [1993] estimate a productivity

improvement of between 3 and 5 using Miranda for an oil reservoir modelling application. Armstrong [1993] estimates that the use of Erlang for real-time telecommunications systems increased productivity, over the complete software cycle, from specification to tested code, by a factor of between 9 and 25!

However the increased productivity does not come for free. One must expect the high level of abstraction to:

- Impose additional execution overheads when the program is run. (This is an instance of the development vs. execution cost trade-off.)
- Reduce the predictability of the execution behaviour.

The extent of these "costs" in current implementations of lazy functional languages is discussed in the following sections.

3.1.1 Current state-of-the-art performance

Recent years has seen a significant amount of research into the efficient implementation of lazy functional languages. The result has been a dramatic improvement in the performance of these languages. Recent research comparing the use of lazy functional languages with more conventional programming languages, such as C and Fortran, have observed an execution performance differential of between 10 and 30 (Grant et al. [1993]; Kozato & Otto [1993]; Sanders & Runciman [1992]). With active research continuing further performance improvements can be expected. The use of functional languages by the "real world" looks likely to grow, provided appropriate support tools are provided.

3.1.2 Predictability

In spite of the improved performance, lazy functional languages still suffer from unexpected runtime behaviour or "performance bugs". The ease of expression, especially when higher-order functions are used, often obscures the time complexity involved. In addition, evaluation often has implicit space requirements. Stoye [1985], Meira [1985] and Peyton Jones [1987] all discuss the problems of predicting this, presenting examples of programs that are semantically identical, but have very different pragmatic space behaviour.

Since reasoning about the space and time behaviour of lazy functional programs is very complex (Meira [1985]), a more pragmatic approach is to put effort into the provision

of better profiling and debugging tools, leaving the programmer to fix the performance problems identified. The insights gained from the use of the heap profiling tool recently added to the **hbc/lml** compiler certainly support this approach (Kozato & Otto [1993]; Runciman & Wakeling [1993]; Runciman & Wakeling [1992]; Sanders & Runciman [1992]).

3.2 Lazy Profiling is Difficult

The key problem faced in profiling any program is to relate the profiling information gathered about the execution of the program back to the original source code in a well-defined and usable manner. This is difficult to achieve when profiling a high-level language, since it provides abstractions and constructs that are unrelated to the underlying execution engine.

Lazy functional languages are no exception. The very features which they advocate, such as:

- many concise functions,
- polymorphism,
- higher-order functions,
- lazy evaluation, and
- program transformation

pose particular problems to a profiler attempting to map profiling data back to the original source.

Some of the problems that lazy languages pose to profiling are discussed in Runciman & Wakeling [1990]. The issues are addressed here with respect to the source mapping issue identified above.

3.2.1 Many concise functions

Functional programming encourages a style of programming which constructs a program from many small function definitions. This results in a program with a very large number of small pieces of code. For example, the Glasgow Haskell compiler consists of over 36000 lines of Haskell source code containing nearly 3000 function definitions averaging under

10 lines of code each. Reporting profiling information for all of these functions would be very cumbersome. Features which aggregate the profiling data in a meaningful and useful way are essential.

3.2.2 Polymorphism

Polymorphism encourages the re-use of functions in many different contexts. Unfortunately this heavy re-use of functions makes it harder to identify the source of observed execution costs. Suppose we wish to know the cost of the expression:

```
map (g x) l
```

Knowing that the program spent 30% of its time in the function **map** is not particularly helpful, since there may be many applications of **map** in the program.

The solution requires the costs of the calls to **map** to be attributed to its call sites. Statistical inheritance (Section 2.5.1) is unlikely to be suitable as the costs of calls from different call sites are likely to vary greatly. For example, the cost of **map** is dependent on the length of the list argument passed and the demand on the resulting list. Subsuming the costs of these heavily re-used functions (Section 2.5.2) would seem to be a more appropriate inheritance technique. The loss of information about these functions should not be significant as they are not critical to the overall cost structure of the program.

3.2.3 Higher-order functions

Higher-order functions are an integral part of functional languages. Hughes [1989] advocates the provision of generalised, higher-order functions, which can then be specialised with appropriate base functions. They pose problems to the profiler since the actual function being applied may not be known at compile time as it is passed as an argument or extracted out of a data structure.

In the New Jersey SML profiler (Appel, Duba & MacQueen [1988]) each profiled function is responsible for setting the *current function* to itself when it is called. This ensures that its execution costs are attributed correctly, even if it is passed as an argument to a higher-order function. The costs of executing an unprofiled function that is passed as an argument to a higher-order function are subsumed by the higher-order function in which it is applied. In contrast, Clack, Clayman & Parrott [1994] argue that the costs of

applying the unprofiled higher-order argument should be attributed to the function that referenced it, not the function that applies it. We discus the details of their *lexical* profiler in Section 3.3.3.

3.2.4 Lazy evaluation

Lazy evaluation poses the profiler with some particular difficulties.

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

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

Finally, it is essential that the lazy semantics are not modified by the profiler. In strict languages one might measure the time taken to execute between two "points" in the source (see Section 2.3.3). However in a lazy language there is no linear evaluation sequence so we no longer have a clear notion of a "point" in the execution. One could imagine a crude profiling scheme that forced the evaluation of the intermediate data structure after each phase of (say) a compiler. This would enable the cost of the each phase to be measured, but we would be measuring the cost of a different program — one that forces its intermediate data and may be evaluating parts which need never be evaluated!

3.2.5 Program transformation and optimisation

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

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

It is highly undesirable to turn off these optimisations, because the resulting profile would not be of the program you actually want to run and improve. Since our aim is to be able to profile a fully optimised program execution, the problem of mapping the costs of *optimised* execution back to the source code must be addressed.

3.3 Lazy Profiling Tools

As noted earlier there are very few profiling systems for lazy functional programs. Typically lazy functional language implementations have only provided basic statistics about the execution of the particular abstract machine and the performance of the storage management system. These implementation statistics do nothing to aid the programmer in identifying the source of any performance problem with their program.

Aside from the **ghc** profiler described in this thesis, I am aware of only three execution profilers for lazy functional programs that relate the profiling data back to the program source, and only one of these profiles execution time.

- The **hbc/lml** heap profiler developed by Runciman & Wakeling [1993].
- The **nhc** heap profiler developed by Röjemo [1994].
- The UCL lexical profiler developed by Clack, Clayman & Parrott [1994].

3.3.1 Hbc/lml heap profiler

Runciman & Wakeling [1993] have implemented a heap profiling scheme for the Chalmers **hbc/lml** compiler (Augustsson & Johnsson [1989]). They map the heap objects back to

the source code by storing, in every heap object, the *function*, *module* and *group* that produced the heap object and the name of the *construction* and *type* of the heap object. The profiling output consists of a graphical display of the contents of the heap over time broken down by function, module, group, construction or type. In addition the programmer can focus the profiling output by limiting the profile to a subset of the heap objects. This selection can be made by any of the function, module, group, construction or type attributes.

Interpreting the heap profiles of particular programs has revealed interesting phenomena about their space behaviour. These insights have led to significant improvements being made to many of the programs that have been profiled (Kozato & Otto [1993]; Runciman & Wakeling [1993]; Runciman & Wakeling [1992]; Sanders & Runciman [1992]) as well as changes to the evaluation scheme used by the compiler itself (Runciman & Wakeling [1993]).

Unfortunately the Runciman and Wakeling profiler does not provide a mechanism for aggregating information up the call graph. A producer profile may indicate that cells produced by a certain function, e.g. `map`, occupy a large amount of heap space. However there is no mechanism to determine which application(s) of `map` were responsible for producing these cells (Kozato & Otto [1993]).

The emphasis of this profiling tool is on the identification and removal of the excessive *space* requirements that lazy functional programs are particularly prone to. Though improving the space behaviour of a program reduces the paging and garbage collection costs, the effect of these changes on the evaluation time is often minimal because most of a program's execution time is usually spent evaluating expressions, not in the garbage collector. Unblocking pipelines and modifying definitions to change strictness properties do not necessarily result in algorithmic changes that reduce the evaluation time. The heap profiles do not provide an explicit indication of the amount of time being consumed by the program parts.

However, if the memory requirements exceed the physical memory of the machine the paging overheads can be quite significant (assuming a virtual memory system is available) (Sansom & Peyton Jones [1993]). Under these circumstances reducing the space requirements can result in substantial performance improvements.

This heap profiler was the first practical profiling tool developed for lazy functional

programs. Its success has a lot to do with the fact that it was incorporated into a widely-used, production-strength compiler, rather than existing as an obscure research toy.

3.3.2 Nhc heap profiler

Nhc (**n**early a **h**askell **c**ompiler) is a light weight compiler for a subset of Haskell developed by Röjemo [1994]. It includes a heap profiler similar to that provided by the **hbc/lml** compiler. However **nhc** incorporates two new heap profiles:

- The *lifetime* profile displays the (selected) heap objects broken down by the length of time each heap object lived. Every heap object has a word that records the creation time of the object with the lifetimes being deduced by post-processing a profile log.
- The *retainer* profile attempts to answer the question: *What is retaining the objects in the heap?* It displays the (selected) heap objects broken down by the (set of) heap objects that reference the object. This profile has been developed as an aid to identifying the cause, rather than the presence, of an unexpected space leak.

These new heap profiles look very promising. They have already provided additional insights into the space-behaviour of the `clausify` program (see Section 6.1) which was initially profiled by Runciman & Wakeling [1993] using the **hbc/lml** profiler (Runciman & Röjemo [1994]).

The **nhc** compiler is still in the early stages of development. It does not profile execution time and does not perform any significant program optimisations.

3.3.3 UCL lexical profiler

Clack, Clayman & Parrott [1994] have implemented a profiling scheme in an interpreted lazy graph-reduction system that profiles call counts, heap usage and execution time of identified functions. Each profiled function is assigned a unique "colour"[1]. The time and space costs of evaluating all expressions declared within the *lexical* scope of the function are attributed to the colour assigned to the function. Though there are encouraging similarities between the UCL profiler and our lexical profiler (see Section 4.2.4), there are also some significant differences.

[1]The UCL notion of "colour" is similar to our "cost centre" — both are attributed with the costs identified during execution.

Subsuming function costs

The UCL profiler requires all shared functions to be profiled separately. Only the costs of unshared, unprofiled functions are subsumed by the referencing function (Section 2.5.2). In contrast, our lexical profiler requires all but only CAFs to be profiled separately (Section 4.1.7). All unprofiled function costs are subsumed by the referencing cost centre (Section 4.1.4). We believe this to be a major strength of our profiling scheme since it enables the costs of the logical "parts" of a program to be aggregated together, regardless of the sharing properties of the program. This is especially important when profiling large applications.

It is important to note that the sharing property, on which the inheritance property of the UCL profiler depends, is global. This is not a problem for an interpreted implementation since this is easy to determine once the code has been loaded into the interpreter. However, in a module-based, compiled implementation the linker has to be modified to mark functions as being shared or unshared, greatly reducing the portability of any implementation. This is not a problem with our approach since the inheritance property is determined locally, by the form of the declaration.

We observe that it would be quite easy to modify the UCL profiler to enable all unprofiled function costs to be subsumed, regardless of the sharing property. All that is required to enable all unprofiled function costs to be subsumed is for the instantiation of unprofiled supercombinators to assign the constructor colour, as well as the origin colour, from the referencing colour pair (see Clack, Clayman & Parrott [1994], Section 5.3). Our experience suggests that this would be a very worthwhile enhancement.

Higher-order functions

To ensure that the colouring of the reference to an unprofiled higher-order function argument is available when the function is applied the UCL profiler attaches colouring information to every field in a closure, as well as the closure itself. This introduces quite a large space overhead. Most of the time the field colouring is redundant since the closure being referenced has the same colouring. Our implementation avoids attaching cost centres to the closure fields. A simple "boxing" transformation is used to ensure that any top-level, unprofiled functions that are passed as arguments have the referencing cost

centre attached (see Section 4.2.4).

Colour pairs

The colouring information recorded by the UCL profiler identifies colour pairs $c \leftarrow o$: the *c*onstructor colour identifies the function being evaluated and the *o*rigin colour identifies the function that referenced it. This enables a more detailed profile to be produced, providing enough information for statistical inheritance (though this has not been implemented). Our current implementation only produces a flat cost centre profile. We rely on the subsuming of all unprofiled function costs. Inheritance profiling using cost-centre pairs is discussed in Section 8.4.2.

Time profiling

The UCL implementation measures execution time by interrogating the system clock whenever the colour of the expression being evaluated changes; recording the elapsed time attributed to the previous colour. The accuracy of this approach is dependent on the accuracy of the system clock, the overhead of accessing it, and the number of times the clock must be accessed.

Accessing the system clock whenever the colour changes imposes an overhead that is inversely proportional to the length of the interval. The shorter the timed intervals the larger the overhead. Under lazy evaluation the time intervals are often very short, especially if the implementation is efficient, since the evaluation of one colour is often interleaved with the evaluation of its inputs. There is also no guarantee that the timing overhead is linear since the intervals which make up execution of one cost centre may be of a different length to the intervals which make up the other cost centres. In addition, most Unix clocks only have a resolution of about 20ms. This implies that the measured time "jumps" in 20ms ticks.

Our implementation attributes execution time by sampling the current cost centre at regular intervals (every 20ms) during the execution. This avoids distorting the profile since the timing overhead is linear (a fixed sampling overhead for each 20ms execution). The statistical variation introduced by the sampling mechanism is no worse than timing a clock with a 20ms tick.

Current status

The UCL implementation is still in a prototype stage with the current version profiling the interpreted execution of the FLIC intermediate code produced by their Haskell compiler. This has a number of acknowledged shortcomings:

- It profiles interpreted execution, not full-blooded compiled code. They are currently working on a compiled TIM implementation.
- The profiling information is related back to the FLIC intermediate code, not the original Haskell source. They do not address the compiler transformation/optimisation issues.

3.3.4 Monitoring semantics

A completely different approach is taken by Kishon [1992]. Kishon introduces the notion of a *monitoring semantics* that is used to specify source-level debuggers, tracers and profilers, for both strict and non-strict languages. A non-standard interpreter for the monitoring semantics is then combined with a standard interpreter for the language to produce a monitored interpreter. Partial evaluation techniques are used to produce a more efficient implementation.

This is a very promising approach for the development of debuggers. However, it is less attractive for practical profilers since it is monitoring the semantics, not the execution. Our profiler attempts to monitor the compiled execution, attributing the "real" costs back to the program source.

Chapter 4

Profiling with Cost Centres

Our profiling system specifically addresses the crucial problem of attributing the profiling data gathered during execution back to the original source code. This is achieved by:

1. Associating expressions of interest in the original source with a *cost centre.*
2. Preserving this association during the transformation and optimisation phases of the compiler.
3. At runtime, identifying the *cost centre* associated with the expression currently being evaluated.
4. Attributing profiling data gathered during execution to the *cost centre* identified.

A *cost centre* is simply a label to which we attribute execution costs. Each cost centre is *attributed with*[1] the costs of evaluating the expression it identifies.

The association of an expression with a cost centre is made very explicit by extending the syntax of expressions with an `scc` (set cost centre) construct.

$$expr \rightarrow \texttt{scc}\ label\ expr$$

This expression-level annotation is very general. It can be used to annotate the entire body of a function or a particular branch of a `case`.

Semantically, an `scc` expression simply returns the value of *expr*, but operationally, it attributes the cost of evaluating *expr* to the cost centre *label.* For example:

[1] In English we would usually say "the cost A is *attributed to* cost centre B. However, since this is really a relation, we often find it more convenient to say "cost centre B is *attributed with* the cost A".

```
mapg x l = scc "mapg" map (g x) l
```

causes the costs of evaluating the expression `map (g x) l` to be attributed to the cost centre `"mapg"`. The syntax for `scc` uses a very loose binding, extending all the way to the right within the enclosing language construct. The scope of an `scc` annotation is restricted by placing brackets around the `scc` annotation. For example:

```
mapg x l = (scc "map" map) (g x) l
```

only annotates the reference to `map`.

For the profiling data collected to be useful, we must provide the programmer with a clear understanding of what costs are attributed to each cost centre. This requires us to define what we mean by *the cost of a source program expression* — its *cost semantics* — and to identify the cost centre to which these costs are attributed. Ideally the necessary understanding should correspond with the programmer's intuition. There should be only a few concepts with which the programmer needs to be familiar to use the profiler and preferably no unexpected pitfalls.

We first identify the desired principles of cost attribution (Section 4.1), before making this precise using a high-level reduction semantics augmented with a notion of cost and cost attribution (Section 4.2).

4.1 Principles of Cost Attribution

In Section 3.2 we identified a number of properties of lazy functional languages that make the task of attributing costs back to the original source difficult. In response to these we have developed a profiling tool which subscribes to the following principles:

- We profile the *actual evaluation* required during normal execution — the evaluation sequence is not changed (Section 4.1.1).
- The costs of evaluating all the *instances* of an `scc`-annotated expression are attributed to its cost centre (Section 4.1.2).
- The costs of evaluating any unevaluated inputs to an expression, i.e. the free variables, are attributed to the declaring scope, not the demanding scope (Section 4.1.3).

- Aggregation of costs is achieved by arranging for the costs of any unprofiled expressions to be *subsumed* (Section 4.1.4).
- Costs are attributed to precisely one cost centre — there is no inheritance of profiled sub-expression costs (Sections 4.1.5 and 4.1.6).
- The one-off costs of evaluating global updateable closures are attributed to special cost centres (Section 4.1.7).

Each of these design decisions is discussed in the following sections.

4.1.1 Degree of evaluation

In a lazy language the extent to which an expression is evaluated depends on the demand placed by the surrounding context. For example, consider the result of the `map` in the expressions

```
sum_mapg  x l = sum      (scc "mapg" map (g x) l)
take_mapg x l = take 10 (scc "mapg" map (g x) l)
```

In `sum_mapg` all the elements of the list are demanded if the result of `sum_mapg` is ever demanded. In `take_mapg` at most `10` elements of the list are required, but the actual number of elements demanded still depends on the number of elements required by the context in which `take_mapg` is called.

The profiler should not affect the degree of evaluation or the evaluation sequence at all. The profiler should measure the cost of evaluating an expression to the extent that is actually required by the program being executed. Let's call this degree of evaluation the *actual evaluation.*

This unknown degree of evaluation results in a potential source of confusion: a programmer might be expecting to measure evaluation that never occurs. However, since we are interested in identifying the critical expressions within the program, we are not concerned with potentially inefficient expressions that are never actually evaluated. If (and only if) the evaluation is demanded, its cost will be measured.

4.1.2 Expression instances

Many *instances* of a single source expression may be evaluated during the execution of a program. For example, an instance of the expression `map (g x) l` in the body of

```
mapg x l = scc "mapg" map (g x) l
```

is evaluated each time `mapg` is called with two arguments. It is not feasible or desirable to report individual costs for every instance of the annotated expression `map (g x) l`. Instead the profiler attributes the cost of evaluating *all* the instances of the `scc` expression to its associated cost centre.

In fact it is possible for more than one `scc` annotation to have the same label. The cost of evaluating all the instances of `scc` expressions with the same label are attributed to a single cost centre with that label.

The number of instances of each `scc` *label* annotation that are evaluated during execution is counted and reported along with the total cost. This is called the "`scc` entry count". It is equivalent to the *entry count* or *frequency count* reported in conventional profiling systems (see Section 2.3.1). Care must be taken if this count is used to average the total cost since the cost incurred by each instance may differ. This is especially true in a lazy language, since the actual evaluation is dependent on the demanding context.

4.1.3 Evaluation of inputs

Under lazy evaluation an expression instance is evaluated only when required; subsequent demands "see" the evaluated form as the expression is updated with its result. The cost of demanding an expression's inputs or free variables therefore depends on the existing degree of evaluation of these inputs. Consider the following definition of `avg`

```
avg l = (scc "sum" sum l) / (scc "len" length l)
```

At most one of the annotated expressions will have to evaluate the spine of `l`, the other will find that the spine of the input list `l` has already been evaluated.

When examining the cost of a particular expression we don't want the water to be muddied by the degree of evaluation of the inputs. We avoid this confusion by excluding from the cost of an expression the cost of evaluating the values bound to its free variables, even though this evaluation occurs interleaved with that of the demanding expression.

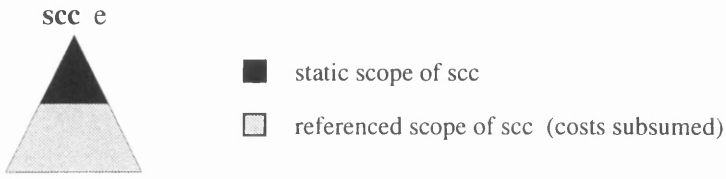


Figure 4.1: Subsumed `scc` scope

We arrive at the same conclusion when we observe that one expression's input is another expression's result. When evaluating the result of a suspended computation the costs should be attributed to the cost centre of the suspension, not to the expression demanding the result.

In the `avg` example above the costs of evaluating `l` should be attributed to the scope responsible for constructing `l`, not to `"sum"` or `"len"`. This corresponds to the intuition we have for strict languages where the evaluation of all inputs to an expression is completed before we evaluate the expression.

4.1.4 Subsuming unprofiled costs

Aggregation of profiling data is very important when profiling functional programs as they typically comprise many small function definitions (Section 3.2.1) which may be heavily re-used (Section 3.2.2). Our profiler arranges for unprofiled costs to be *subsumed* by the caller (Section 2.5.2). This enables the programmer using the profiler to accurately:

- Determine the total cost of a (possibly large) nest of function calls.
- Attribute the costs of heavily re-used function definitions to the cost centres of their application sites.

All top-level functions are considered to be unprofiled — though they may have profiled expressions embedded within. Indeed, the entire body of the function may be a profiled expression. The costs of evaluating any unprofiled expressions within a top-level function are *subsumed* by the expression that *referenced* the function, just as if the top-level function had been *unfolded at the site where the function is referred to.* This subsumed scope is depicted in Figure 4.1. In the example

```
mapg x l = scc "mapg" (map (g x) l)
```

the evaluation of map and the evaluation of all the applications of g hidden inside map are attributed to "mapg". Any other applications of map are attributed to the cost centre enclosing that application site.

We use the term *reference site* to identify the source location where the function is referred to. This may be different from the *application site* since the function may be passed as an argument (to a higher order function) and applied at a different site, possibly in the scope of a different cost centre. For example, in the declaration:

```
app f = scc "app" f 1
ref   = scc "ref" app sum
```

sum is referenced in the scope of "ref", but is applied in the scope of "app". Any costs associated with the evaluation of sum should be attributed to the referencing cost centre "ref", not the applying cost centre "app".

4.1.5 Profiled sub-expressions

A profiled expression may have a profiled sub-expression embedded within it. This might arise from an explicit scc sub-expression or an scc expression embedded within a subsumed top-level function. Consider the expression:

```
sum_mapg x l = scc "sum" sum (scc "mapg" map (g x) l)
```

Should the cost of the map be attributed to the cost centre "sum" as well as "mapg"? We adopt a very simple scheme: *Costs are only attributed to a single cost centre.* The cost of the inner expression, map (g x) l, is attributed to the cost centre "mapg" and the cost of summing the result is attributed to "sum".

Therefore the scope of a particular cost centre may have "holes" in it that correspond to annotated sub-expressions which attribute their costs to another cost centre (see Figure 4.2). So that the existence of any annotated sub-expressions does not go unnoticed by the programmer, we count the number of sub-scc expression instances evaluated. When scc "mapg" is entered in the example above, the scc entry count of the cost centre "mapg" and the sub-scc count of "sum" would be incremented. The sub-scc count does not identify which cost centre(s) the sub-scc expression(s) are attributing their costs to — just that there are such expressions.

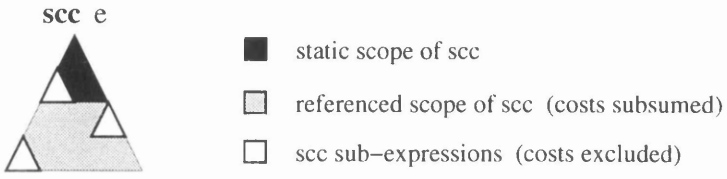


Figure 4.2: The scope of an `scc` expression

4.1.6 Inheritance and profiled sub-expressions

It is quite possible to take a different approach to that of 4.1.5, and arrange for the costs of profiled sub-expressions to be attributed to the enclosing cost centre(s) as well. This is equivalent to the call graph profiling described in Section 2.5.1 except that we would use the *reference graph.*

However, collecting accurate inherited information is very expensive. We would have to keep track of the reference stack of cost centres for *every* unevaluated expression. (The explicit stack in a lazy implementation is a demand stack.)

On the other hand, statistical inheritance is feasible. It requires the runtime costs to be attributed to *cost-centre pairs.* This is discussed in Section 8.4.2.

Given that we already have accurate subsuming of unprofiled costs, providing a form of cost aggregation, statistical inheritance was not a high priority. We decided not to implement it in the initial implementation, prefering to concentrate our effort on the more fundamental problems identified. However, we do believe that combining statistical inheritance with the subsuming of unprofiled costs could prove to be a useful extension to the profiling tool.

4.1.7 Global updateable closures (CAFs)

Section 4.1.4 stated that all top-level functions are considered to be unprofiled — their costs are subsumed by the reference site. However some top-level closures may have no arguments, and hence be updateable. They are only evaluated once (if at all), and only to the extent to which they are demanded. These argument-less top-level closures are called *constant applicative forms* or CAFs. For example, `ints` is a CAF whose value is the infinite list of integers:

```
ints = from 0
```

where `from` is a function returning the infinite list of integers starting from its argument.

CAF cost centres

Since each CAF is only evaluated once, the one-off costs of evaluation should be attributed to the declaration site of the CAF (Section 4.1.3). Otherwise, these costs would be attributed to the cost centre of the first expression to demand the value of the CAF. Understanding which cost centre was attributed with the evaluation of the CAF would require the programmer to reason about the evaluation order.

So to which cost centre should these costs be attributed? To ensure that we always have a cost centre to which to attribute these costs the compiler annotates every CAF with an `scc` annotation (if it doesn't already have one). By default a single `"CAF"` cost centre label is used to annotate all CAFs in a module, but a compiler option is provided that instructs the compiler to annotate each CAF with a cost centre derived from the name of the CAF. For example, the individual annotation for `ints` would be:

```
ints = scc_caf "CAF:ints" from 0
```

Entry to CAF `scc` expressions is also treated specially. We know that there is only one instance of each CAF expression, but we do not know which reference to the CAF will be the first to demand its value and force its evaluation. If we increment the sub-`scc` count of the first expression to demanding evaluation of the CAF, the sub-`scc` count would be dependent on the evaluation order. To avoid this undesirable outcome, $\texttt{scc}_{caf}$ does not increment the sub-`scc` count of the demanding cost centre. Instead we increment a sub-$\texttt{scc}_{caf}$ count which we know is dependent on the evaluation order.

Non-updated CAFs

In an effort to save space it is possible to arrange for CAFs (with large results) to be re-evaluated every time their value is required. These repeated evaluation costs could be inherited by the expression demanding the result. However, the compile time (or runtime) decision to avoid updating a top-level thunk and pay the cost of any re-evaluation to save space should not affect the costs attributed to the demanding cost centre(s). Thus we

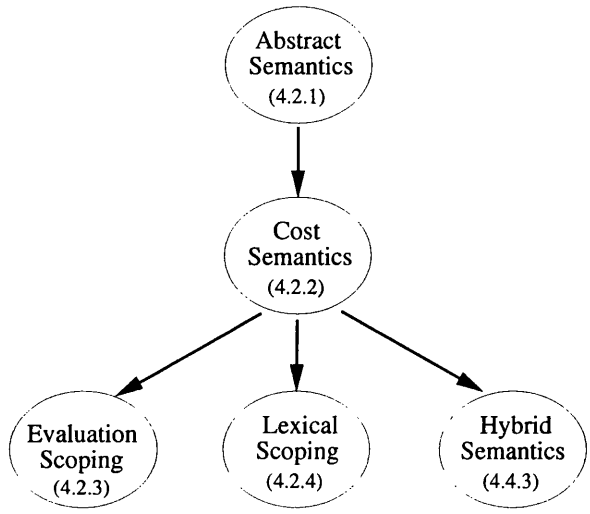


Figure 4.3: Development of abstract cost semantics

still annotate these CAFs and attribute the cost of repeated evaluation to their CAF cost centre. The CAF scc entry count records the number of instances of the CAF which are evaluated during execution.

4.2 Abstract Cost Semantics

Our initial profiling implementation was based on the informal principles developed in Section 4.1. However, some very subtle issues emerged that were difficult to identify and investigate in such an informal setting. This led us to develop a more formal notion of cost attribution that enables us to be precise about these issues.

In order to explain precisely how costs are attributed it is necessary to reason about the operational behaviour of the program. To this end we introduce an abstract reduction semantics, which we then extend with notions of cost and cost attribution. These cost semantics are sufficiently concrete to allow us to be precise about the evaluation behaviour and cost attribution, but are sufficiently abstract that we do not get bogged down in irrelevant details.

The development of the cost semantics is summarised in Figure 4.3. Initially two abstract cost semantics, *lexical scoping* and *evaluation scoping*, are developed and compared. Experience gained from the two profiling schemes leads to the development of a third,

hybrid cost semantics.

4.2.1 Abstract reduction rules

The reduction semantics used here are based on Launchbury's natural semantics (Launchbury [1993a]). The semantics are given for the following language:

$$
\begin{array}{lcll}
x & \in & Var & \\
C & \in & Constructor & \supset \quad Numbers \\
\oplus & \in & Primitive & \\
e & \in & Exp & ::= \quad \lambda x.e \\
 & & & \mid \quad e\ x \\
 & & & \mid \quad x \\
 & & & \mid \quad \texttt{let}\ x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n\ \texttt{in}\ e \\
 & & & \mid \quad C\ x_1 \cdots x_n \\
 & & & \mid \quad \texttt{case}\ e\ \texttt{of}\ \{C_i\ x_{1_i} \cdots x_{m_i}\ \texttt{->}\ e_i\}_{i=1}^{n} \\
 & & & \mid \quad e_1 \oplus e_2
\end{array}
$$

This language contains a minimal set of constructs required to implement Haskell without losing any efficiency. It consists of the lambda calculus extended with (recursive) lets, saturated constructors (including numbers), case, and primitive applications[2].

The language also contains an important syntactic restriction: all function and constructor applications must have variables as arguments. This is easily achieved by `let`-binding any non-variable arguments. It forces all closure allocation to be made explicit (the `let` construct is the only construct that allocates closures in the heap) giving the language a more direct operational reading.[3]

In presenting the semantics we assume that all bound variables are distinct. This is ensured by renaming all the bound variables in an expression with fresh variables, written $\widehat{e}$, whenever an expression is duplicated.

The dynamic semantic rules are presented in Figure 4.4. They obey the following conventions. The heap is a partial mapping from variables to expressions. It is viewed as an (unordered) set of variable/expression pairs, binding distinct variables to expressions.

$$\Gamma, \Delta, \Theta \in Heap ::= \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$$

[2] We assume that the primitive operators $\oplus$ (e.g. +, *, ==, < etc.) are strict in both arguments and return a nullary constructor, such as a number or boolean.

[3] An even more restricted language is used in Appendix A to present a direct operational semantics for the STG-machine.

Γ_{init} is the initial heap, binding all the variables declared at the top-level. A value z is an expression in *weak head normal form* (whnf) i.e. a lambda abstraction or saturated constructor application.

$$\begin{array}{rcll} z & \in & Val & ::= & \lambda x.e \\ & & & \mid & C\ x_1 \cdots x_n \end{array}$$

A judgement has the form $\Gamma : e \Downarrow \Delta : z$ which should be read: "the term e in the context of the set of bindings Γ reduces to the value z together with the (modified) set of bindings Δ." During the course of evaluation new bindings may be added to the heap and old bindings updated with their results.

Reduction rules

Referring to the rules in Figure 4.4, the *Lambda* and *Constructor* rules simply reduce lambda abstractions and constructor applications to themselves, without affecting the heap. Such terms are already in whnf so have no need for further evaluation.

The *Application* rule reduces the term on the left (to a λ-abstraction), substitutes the argument for the λ-variable, and continues reduction. Since the syntax ensured that all application arguments are variables no work is duplicated by the substitutions.

The most interesting rule is the *Variable* rule. To evaluate a variable x the heap must contain a binding of the form $x \mapsto e$. Assuming it does, e is evaluated in the context of the heap, *omitting the reference to* x. This ensures that any cyclic data dependencies, or black holes, are detected. If this reduction produces a value z a renamed version of the result $\widehat{z}$ is returned. This renaming ensures that no name clashes occur as a result of duplicating the resulting term z. If the original expression e was not in whnf (captured by the WHNF selector) the heap is updated with a binding $x \mapsto z$, otherwise the original whnf binding is simply restored. The update ensures that subsequent references to x immediately return the result value z.

The conditional update, defined using the WHNF selector, is not actually required for the abstract semantics. It would suffice to always update a binding with its result. We have gone to the trouble of identifying the case when no update is required because this is significant when the costs of evaluation are considered (Section 4.2.2).

The remaining rules are quite straight forward. The *Let* rule extends the heap with the

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textit{Lambda}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : z}{\Gamma : e\ x \Downarrow \Theta : z} \qquad \textit{Application}$$

$$\frac{\Gamma : e \Downarrow \Delta : z}{\{\Gamma, x \mapsto e\} : x \Downarrow \{\Delta, \text{WHNF}(e, x \mapsto e, x \mapsto z)\} : \widehat{z}} \qquad \textit{Variable}$$

where
$$\begin{aligned} \text{WHNF}(\ \lambda x.e, n, u) &= n \\ \text{WHNF}(C\ x_1 \cdots x_n, n, u) &= n \\ \text{WHNF}(\ e, n, u) &= u \end{aligned}$$

$$\frac{\{\Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\} : e \Downarrow \Delta : z}{\Gamma : \texttt{let}\ x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n\ \texttt{in}\ e \Downarrow \Delta : z} \qquad \textit{Let}$$

$$\Gamma : C\ x_1 \cdots x_n \Downarrow \Gamma : C\ x_1 \cdots x_n \qquad \textit{Constructor}$$

$$\frac{\Gamma : e \Downarrow \Delta : C_k\ x_1 \cdots x_{m_k} \quad \Delta : e_k[x_i/y_i]_{i=1}^{m_k} \Downarrow \Theta : z}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{C_i\ y_1 \cdots y_{m_i}\ \texttt{->}\ e_i\}_{i=1}^{n} \Downarrow \Theta : z} \qquad \textit{Case}$$

$$\frac{\Gamma : e_1 \Downarrow \Delta : z_1 \quad \Delta : e_2 \Downarrow \Theta : z_2}{\Gamma : e_1 \oplus e_2 \Downarrow \Theta : z_1 \oplus z_2} \qquad \textit{Primitive}$$

Figure 4.4: Dynamic Semantic Rules

new bindings and evaluates the body e. Renaming ensures that there are no name clashes. *Case* reduces the body (to a constructor), and then reduces the appropriate alternative after substituting the constructor arguments returned. The *Case* rule only succeeds if the constructor returned is contained in the alternatives. Finally, the *Primitive* rule evaluates each argument (left to right) and returns the result of applying the primitive operator.

Reduction sequences

Reduction sequences are expressed using proof trees. To stress the sequential nature of reduction we lay these proofs out vertically. If $\Gamma : e \Downarrow \Delta : z$ we write:

$\{\Gamma\} : e$
| a sub-proof
| another sub-proof
$\{\Delta\} : z$

with sub-derivations (proving the judgements above the line) contained within the vertical bars. For example, the reduction sequence for the expression `let` $f{=}\lambda x.x{+}1$ `in` $f\ 3$ would be written:

$\{\Delta\} : \texttt{let}\ f{=}\lambda x.x{+}1\ \texttt{in}\ f\ 3$ — *Let*
| $\{\Delta, f \mapsto \lambda x.x{+}1\} : f\ 3$ — *Application*
| | $\{\Delta, f \mapsto \lambda x.x{+}1\} : f$ — *Variable*
| | | $\{\Delta\} : \lambda x.x{+}1$ — *Lambda*
| | | $\{\Delta\} : \lambda x.x{+}1$
| | $\{\Delta, f \mapsto \lambda x.x{+}1\} : \lambda x_1.x_1{+}1$ — (no update)
| | $\{\Delta, f \mapsto \lambda x.x{+}1\} : 3{+}1$ — *Primitive*
| | | $\{\Delta, f \mapsto \lambda x.x{+}1\} : 3$ — *Constructor*
| | | $\{\Delta, f \mapsto \lambda x.x{+}1\} : 3$
| | | $\{\Delta, f \mapsto \lambda x.x{+}1\} : 1$ — *Constructor*
| | | $\{\Delta, f \mapsto \lambda x.x{+}1\} : 1$
| | $\{\Delta, f \mapsto \lambda x.x{+}1\} : 4$ — (evaluate +)
| $\{\Delta, f \mapsto \lambda x.x{+}1\} : 4$
$\{\Delta, f \mapsto \lambda x.x{+}1\} : 4$

(combine)

Evaluating a variable that is already in whnf requires two reduction steps. As this is such

a common operation we will combine the two steps required into a single step. This is a notational convenience intended to reduce the length of (tedious) reduction sequences.

4.2.2 Cost augmented reduction rules

We now extend the reduction semantics with a notion of cost and cost attribution. The intention is to precisely identify the costs that are attributed to each cost centre. We first add a new language construct, scc, that associates a cost centre, cc, with the evaluation of an expression e:

$$\begin{array}{rcll} cc & \in & \mathit{CostCentre} & \\ e & \in & \mathit{Exp} & ::= \ \mathtt{scc}\ cc\ e \\ & & & \mid \ \vdots \end{array}$$

The dynamic semantics are then extended with each reduction rule reporting the costs attributed to each cost centre. The cost attribution θ is represented as a partial mapping from cost centres to integers. The costs of two attributions can be combined using $\uplus$ which determines the total cost attributed to each cost centre.

$$\begin{array}{rcl} \theta \in \mathit{Attribution} & = & \{cc_1 \mapsto n_1, \ldots, cc_k \mapsto n_k\} \\ \\ \theta(cc) & = & n_i \ \textbf{if}\ cc = cc_i \\ & = & 0 \ \textbf{otherwise} \\ \\ (\theta_1 \uplus \theta_2)(cc) & = & \theta_1(cc) + \theta_2(cc) \end{array}$$

In addition variable/expression pairs in the heap are annotated with the cost centre associated with the declaration site of the expression.

$$\Gamma, \Delta, \Theta \ \in \ \mathit{Heap} \ ::= \ \{x_1 \overset{cc_1}{\mapsto} e_1, \ldots, x_n \overset{cc_n}{\mapsto} e_n\}$$

The costs of evaluating these heap-bound expressions are attributed to the annotating cost centre. This ensures that the evaluation of the inputs to an expression is attributed to the declaration site, not the demanding expression (Section 4.1.3).

The judgement form is extended to $cc, \Gamma : e \Downarrow_\theta \Delta : z, cc_z$ that should be read: "the term e in the context of the set of (annotated) bindings Γ and enclosing cost centre cc, reduces to the value z together with the (modified) set of (annotated) bindings Δ and result cost centre cc_z, attributing costs θ." The result cost centre, cc_z, is the cost centre

that enclosed the expression that declared or constructed the result value z. The need to return this cost centre will be explained later.

The initial top-level heap bindings, Γ_{init}, depend on the form of expression bound.

- Function values are annotated with the special cost centre "SUB". Since the costs of top-level functions are subsumed by the reference site this cost centre is never associated with an expression during evaluation.
- CAFs are annotated with the cost centre "CAF". The costs of evaluating all CAFs are attributed to the "CAF" cost centre. This corresponds to the explicit scc annotation of CAFs described in Section 4.1.7.

$$\begin{aligned}\Gamma_{init} &= \{x_1 \overset{cc_1}{\mapsto} e_1, \ldots, x_m \overset{cc_m}{\mapsto} e_m\} \\ &\quad \textbf{where } cc_i = \text{"SUB"} \quad \textbf{if } e_i == \lambda x.e \\ &\qquad\qquad = \text{"CAF"} \quad \textbf{otherwise}\end{aligned}$$

Finally, we introduce the following constant costs which are intended to reflect the costs of the reduction steps in a particular implementation:

R_λ:	the cost of returning a lambda abstraction.
R_C:	the cost of returning a constructor.
H:	the cost of allocating a closure in the heap.
V:	the cost of evaluating a variable.
U:	the cost of an update.
A:	the cost of a curried application.
C:	the overheads of a case expression.
P:	the cost of a primitive application.

The cost attribution for each reduction rule is derived from the attribution reported by any sub-reductions and the costs associated with the reduction rule itself.

The value of these abstract costs may vary greatly between different implementations. However, the actual or relative size of these costs is not particularly important to the cost attribution semantics since they are not reported to the user: instead, the implementation measures and reports actual execution time rather than these abstract costs. What *is* important is the cost centre to which each cost is attributed. The abstract costs above enable us to specify and reason about different cost attribution models, the effect of

different program transformations on the attribution of costs, and the correctness of the implementation.

Reduction rules

The augmented reduction rules are given in Figure 4.5. Apart from the addition of cost centres and attribution information the rules are identical to the those given earlier in Figure 4.4. The evaluation semantics have not been modified at all. The same expressions are evaluated in the same order.

The only new reduction rule is the rule for **scc** annotations. It evaluates the annotated expression e in the context of the annotating cost centre cc_{scc}. The cost attribution reported will reflect the fact that e is evaluated in the scope of cc_{scc}. There is no fixed cost associated with an **scc** reduction as it is not part of normal execution. Though it is not specified as part of the semantics, this reduction should also increment the **scc** entry count of cc_{scc} and the sub-**scc** count of cc.

The *Lambda* and *Constructor* rules return the enclosing cost centre with the value — this is the cost centre which encloses the scope that declared/constructed the result. The cost of returning the value is attributed to this cost centre.

The interesting rule in the cost semantics is the *Application* rule. It reduces the term on the left, substitutes the argument for the λ-variable, and reduces the body of the λ-abstraction, e'. The question is: *Where should the cost of evaluating the body of the λ-abstraction be attributed:*

cc: the cost centre enclosing the application, or

cc_λ: the cost centre of the λ-abstraction?

The (possible) use of cc_λ, the cost centre returned with the λ-abstraction is the reason for introducing a cost centre attached to the the result of a judgement. We explore this decision in Sections 4.2.3 and 4.2.4.

The *Variable* rule encounters a heap binding with the form $x \overset{cc_e}{\mapsto} e$. It evaluates the bound expression e, attributing the costs to the cost centre cc_e (the cost centre annotated by the declaration site) unless this is a "SUB" cost centre (attached to top-level functions in Γ_{init}). In this case the costs are attributed to the demanding cost centre cc. This choice is captured in the SUB selector. If the heap is updated the updated binding is annotated

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

$$cc, \Gamma : \lambda x.e \Downarrow_{\{cc \mapsto \mathrm{R}_\lambda\}} \Gamma : \lambda x.e, cc \quad \textit{Lambda}$$

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

$$\frac{\mathrm{SUB}(cc_e, cc), \Gamma : e \Downarrow_\theta \Delta : z, cc_z}{cc, \{\Gamma, x \overset{cc_\xi}{\mapsto} e\} : x \Downarrow_{\theta_{res}} \{\Delta, \mathrm{WHNF}(e, x \overset{cc_\xi}{\mapsto} e, x \overset{cc_\xi}{\mapsto} z)\} : \widehat{z}, cc_z} \quad \textit{Variable}$$

where $\theta_{res} = \{cc \mapsto \mathrm{V}\} \uplus \{cc_z \mapsto \mathrm{WHNF}(e,0,\mathrm{U})\} \uplus \theta$

$$\begin{aligned} \mathrm{WHNF}(\ \lambda x.e, n, u) &= n & \mathrm{SUB}(\texttt{"SUB"}, cc) &= cc \\ \mathrm{WHNF}(C\ x_1 \cdots x_n, n, u) &= n & \mathrm{SUB}(\ cc_e, cc) &= cc_e \\ \mathrm{WHNF}(\ e, n, u) &= u \end{aligned}$$

$$\frac{cc, \{\Gamma, x_1 \overset{cc}{\mapsto} e_1, \ldots, x_n \overset{cc}{\mapsto} e_n\} : e \Downarrow_\theta \Delta : z, cc_z}{cc, \Gamma : \texttt{let}\ x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n\ \texttt{in}\ e \Downarrow_{\{cc \mapsto n*\mathrm{H}\} \uplus \theta} \Delta : z, cc_z} \quad \textit{Let}$$

$$cc, \Gamma : C\ x_1 \cdots x_n \Downarrow_{\{cc \mapsto \mathrm{R}_C\}} \Gamma : C\ x_1 \cdots x_n, cc \quad \textit{Constructor}$$

$$\frac{cc, \Gamma : e \Downarrow_{\theta_1} \Delta : C_k\ x_1 \cdots x_{m_k}, cc_C \qquad cc, \Delta : e_k[x_i/y_i]_{i=1}^{m_k} \Downarrow_{\theta_2} \Theta : z, cc_z}{cc, \Gamma : \texttt{case}\ e\ \texttt{of}\ \{C_i\ y_1 \cdots y_{m_i}\ \texttt{->}\ e_i\}_{i=1}^{n} \Downarrow_{\{cc \mapsto \mathrm{C}\} \uplus \theta_1 \uplus \theta_2} \Theta : z, cc_z} \quad \textit{Case}$$

$$\frac{cc, \Gamma : e_1 \Downarrow_{\theta_1} \Delta : z_1, cc_1 \qquad cc, \Delta : e_2 \Downarrow_{\theta_2} \Theta : z_2, cc_2}{cc, \Gamma : e_1 \oplus e_2 \Downarrow_{\{cc \mapsto \mathrm{P}\} \uplus \theta_1 \uplus \theta_2} \Theta : z_1 \oplus z_2, cc} \quad \textit{Primitive}$$

Figure 4.5: Cost Augmented Dynamic Semantic Rules

with the result cost centre cc_z. This ensures that subsequent references to x also return the cost centre responsible for declaring or constructing the result. The reduction rule has two explicit cost components:

V: the cost of demanding the value of the variable. This is attributed to the demanding cost centre *cc*.

U: the cost of performing the update[4]. If the closure e was not in whnf it must be updated with its result. The cost of the update is attributed to the result cost centre cc_z. No update cost is attributed if the closure was already in whnf as no update is actually required. This is captured using the WHNF selector introduced earlier. Updates are discussed further in Section 5.5.

The *Let* rule extends the heap with the new bindings, annotating each binding with the enclosing cost centre *cc*. The costs of allocating the closures (n*H) are attributed to *cc* and combined with the costs of reducing e.

The *Case* rule reduces the body and the appropriate alternative in the context of the enclosing cost centre *cc*. The cost of the case is also attributed to *cc*. The *Case* and *Application* rules are quite similar: the *Application* rule evaluates a function whose body is then applied to the argument; while the *Case* rule selects an alternative that is "applied" to the arguments of the constructor. However, the choice of cost centre that arose in the *Application* rule does not appear in the *Case* rule. The reason for this can be seen if the implicit "λ-abstraction" and its "application" are made explicit:

$$\texttt{case}\ e\ \texttt{of}\ \{C_i\ y_{1_i} \cdots y_{m_i}\ \texttt{->}\ (\lambda\ x_1 \cdots x_{m_i}.e_i)\ y_1 \cdots y_{m_i}\}_{i=1}^{n}$$

Observe that the implicit λ-abstraction is declared in the same scope as it is applied. Thus there is no choice of cost centre to be made.

Finally, the *Primitive* rule evaluates each argument in the context of the enclosing cost centre *cc* and applies the primitive operator. The enclosing cost centre *cc* is attached to the result — this is the cost centre which encloses the scope that computed and constructed the result $z_1 \oplus z_2$. The cost of the primitive application, P, is attributed to *cc*.

[4]The actual update costs depend on the return/update conventions employed by the actual implementation for the particular result value returned. For example, x may be bound to a copy of the closure, or updated with an indirection to a single, shared copy of the closure.

Reduction sequences

The reduction sequences are extended to include the manipulation of cost centres. If $cc, \Gamma : e \Downarrow_\theta \Delta : z, cc_z$ we write:

$$
\begin{array}{l}
cc, \{\Gamma\} : e \\
\quad \Big| \quad \text{a sub-proof} \\
\quad \Big| \quad \text{another sub-proof} \\
\{\Delta\} : z, cc_z \quad \theta
\end{array}
$$

For convenience, we will omit θ if we are only interested in identifying the cost centre in the context of a particular expression evaluated in the reduction sequence.

4.2.3 Evaluation scoping

The cost semantics described in Section 4.2.2 left an important question to be answered about the *Application* rule: *Where should the cost of evaluating the body of the λ-abstraction be attributed?*

The first profiling semantics we investigate attributes the cost of evaluating the body of the λ-abstraction to *the cost centre enclosing the application site.*

$$
\frac{cc, \Gamma : e \Downarrow_{\theta_1} \Delta : \lambda y.e', cc_\lambda \qquad cc, \Delta : e'[x/y] \Downarrow_{\theta_2} \Theta : z, cc_z}{cc, \Gamma : e\ x \Downarrow_{\{cc \mapsto A\} \uplus \theta_1 \uplus \theta_2} \Theta : z, cc_z} App_{eval}
$$

We use the term *evaluation scoping* to describe this profiling semantics since it corresponds quite closely to the underlying lazy evaluation mechanism (see Section 5.5.6). Each cost centre is attributed with the costs of the *actual evaluation* of the results of all the instances of the annotated expression. The amount of actual evaluation depends on the amount of the result of an expression instance that is demanded by the surrounding program. We term the final form of the result of an expression instance demanded by the *actual evaluation* of the program the *actual normal form* (ANF).

The cost attribution of evaluation scoping can be rather counter-intuitive. Consider the reduction sequence for the expression:

$$\texttt{scc}\ cc_{let}\ \texttt{let}\ h{=}\lambda x.e_h\ \texttt{in}\ \texttt{scc}\ cc_{app}\ h\ l$$

$$\begin{array}{l} cc_{let}, \{\Delta\} : \texttt{let}\ h{=}\lambda x.e_h\ \texttt{in}\ \texttt{scc}\ cc_{app}\ h\ l \\ \quad cc_{let}, \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : \texttt{scc}\ cc_{app}\ h\ l \\ \quad\quad cc_{app}, \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : h\ l \\ \quad\quad\quad cc_{app}, \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : h \\ \quad\quad\quad \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : \lambda x_1.\widehat{e}_h[x_1/x], cc_{let} \\ \quad\quad\quad cc_{app}, \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : \widehat{e}_h[l/x] \\ \quad\quad\quad\quad \vdots \end{array}$$

The function h is declared within the scope of cc_{let}, but the costs of evaluating the body of h, are attributed to the cost centre cc_{app} that encloses its application site. The costs attributed to the cost centre cc_{app} are dependent on the declaration of h which is not within the scope of the cost centre. Modifying the definition of h, which is outside the scope of the scc cc_{app} expression, directly affects the costs attributed to cc_{app}.

However, since the costs of evaluating the body of a function are attributed to the application site we can distinguish between the costs of several applications of the same function, which might be useful. For example, the costs of each application of h in the expression

$$\texttt{scc}\ cc_{let}\ \texttt{let}\ h{=}\lambda x.e_h\ \texttt{in}\ (\texttt{scc}\ cc_{app_1}\ h\ l_1, \texttt{scc}\ cc_{app_2}\ h\ l_2)$$

are attributed to the distinct cost centres, cc_{app_1} and cc_{app_2} that enclose each application site.

4.2.4 Lexical scoping

The alternative to evaluation scoping is to attribute the costs of the application and evaluating of the body of the λ-abstraction to the cost centre attached to the λ-abstraction.

$$\frac{cc, \Gamma : e \Downarrow_{\theta_1} \Delta : \lambda y.e', cc_\lambda \qquad cc_\lambda, \Delta : e'[x/y] \Downarrow_{\theta_2} \Theta : z, cc_z}{cc, \Gamma : e\ x \Downarrow_{\{cc_\lambda \mapsto \mathrm{A}\} \uplus \theta_1 \uplus \theta_2} \Theta : z, cc_z}\ App_{lex}$$

We use the term *lexical scoping*[5] to describe this profiling semantics because it has the

[5]This term was was taken from the UCL profiler terminology (Section 3.3.3). They use this notion of *lexical scope* as the underlying principle in the development of their profiler. Though we have adopted

very appealing property that:

> *The cost of executing all the "code" lexically enclosed within an annotated expression is attributed to the annotating cost centre.*

The cost attribution of lexical scoping is more suitable for profiling because the costs attributed to a cost centre are only dependent on the expression enclosed within the scope of the cost centre. The example reduction sequence under lexical scoping is:

$$\texttt{scc}\ cc_{let}\ \texttt{let}\ h{=}\lambda x.e_h\ \texttt{in}\ \texttt{scc}\ cc_{app}\ h\ l$$

$$\begin{array}{l} cc_{let}, \{\Delta\} : \texttt{let}\ h{=}\lambda x.e_h\ \texttt{in}\ \texttt{scc}\ cc_{app}\ h\ l \\ \quad cc_{let}, \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : \texttt{scc}\ cc_{app}\ h\ l \\ \quad\quad cc_{app}, \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : h\ l \\ \quad\quad\quad cc_{app}, \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : h \\ \quad\quad\quad \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : \lambda x_1.\widehat{e}_h[x_1/x], cc_{let} \\ \quad\quad\quad cc_{let}, \{\Delta, h \overset{cc_{let}}{\mapsto} \lambda x.e_h\} : \widehat{e}_h[l/x] \\ \quad\quad\quad\quad \vdots \end{array}$$

The cost of evaluating the body of h is attributed to the cost centre cc_{let} that encloses the declaration of h. Modifying the definition of h, affects the costs attributed to the declaring cost centre cc_{let}. The cost centre cc_{app} is only attributed with the cost of building and subsequently entering the closure for $h\ l$.

Though this may seem a more intuitive semantics it has the disadvantage that the costs of different application sites cannot be distinguished. In the example expression

$$\texttt{scc}\ cc_{let}\ \texttt{let}\ h{=}\lambda x.e_h\ \texttt{in}\ (\texttt{scc}\ cc_{app_1}\ h\ l_1, \texttt{scc}\ cc_{app_2}\ h\ l_2)$$

cc_{let} is attributed with the costs of both the applications.

The reduction example above is for a λ-abstraction declared in the scope of one cost centre and applied in the scope of another. There is no difference between the profiling schemes if the λ-abstraction is declared and applied in the scope of the same cost centre (since $cc_{let} = cc_{app}$) or if the λ-abstraction is a top-level subsumed function. In this latter case both schemes attribute the costs of evaluating the body of the λ-abstraction to the cost centre enclosing the application site (but see below).

---

their terminology this work was developed independently.

Boxing top-level function arguments

The semantics, as currently specified, do not implement true lexical scoping. Section 4.1.4 requires the costs of applying subsumed functions to be attributed to their reference site. Unfortunately the SUB selector in the *Variable* rule selects the demanding cost centre of the application site and returns this with the λ-abstraction (via the *Lambda* rule). This may be different from the cost centre that enclosed the reference site if the top-level function is passed as a higher-order argument. Consider the reduction sequence:

$$\Gamma = \Delta, sum \overset{\text{"SUB"}}{\mapsto} \lambda xs.e_{sum}, app \overset{cc_{app}}{\mapsto} \lambda f.f\ l$$

$$\begin{array}{l} cc_{ref}, \{\Gamma\} : app\ sum \\ \quad \left| \begin{array}{l} cc_{ref}, \{\Gamma\} : app \\ \{\Gamma\} : \lambda f_1.f_1\ l, cc_{app} \end{array} \right. \\ \quad \left| \begin{array}{l} cc_{app}, \{\Gamma\} : sum\ l \\ \quad \left| \begin{array}{l} cc_{app}, \{\Gamma\} : sum \\ \{\Gamma\} : \lambda xs_1.\widehat{e}_{sum}[xs_1/xs], cc_{app} \end{array} \right. \\ \quad \left| \begin{array}{l} cc_{app}, \{\Gamma\} : \widehat{e}_{sum}[l/xs] \\ \qquad \vdots \end{array} \right. \end{array} \right. \end{array}$$

The top-level function, *sum*, is referenced in the scope of cc_{ref}, but the costs of evaluating the body of *sum* are attributed to the cost centre enclosing the application site, cc_{app}. The problem is that *sum* is substituted in the body of *app*, the function it is passed to, without recording the cost centre of its reference site, cc_{ref}.

The solution requires us to "box" any top-level functions that are passed as arguments with the cost centre of the reference site. When this function is later applied the "boxing" cost centre is returned and the evaluation of the body attributed to it. This is specified with a simple static transformation that `let`-binds any top-level function names being passed as arguments — written $e^{\#}$. The key rules in the transformation are:

$$\begin{array}{lcll} (e\ x)^{\#} & = & \texttt{let } y\texttt{=}x \texttt{ in } (e^{\#})\ y & \textbf{if } x \textbf{ a top-level function } [y \textit{ fresh}] \\ & = & (e^{\#})x & \textbf{otherwise} \\ \\ (C\ x_1 \cdots x_n)^{\#} & = & \texttt{let } y\texttt{=}x_i \texttt{ in } ((C\ x_1 \cdots x_n)[y/x_i])^{\#} & \\ & & & \textbf{if some } x_i \textbf{ a top-level function } [y \textit{ fresh}] \\ & = & C\ x_1 \cdots x_n & \textbf{otherwise} \end{array}$$

The other cases simply apply the transformation to all sub-expressions. The reduction sequence above now becomes:

$$\Gamma = \Delta, sum \overset{\text{"SUB"}}{\mapsto} \lambda xs.e_{sum}, app \overset{cc_{app}}{\mapsto} \lambda f.f\ l$$

$cc_{ref}, \{\Gamma\}$: `let` y=sum `in` $app\ y$

- $cc_{ref}, \{\Gamma, y \overset{cc_{ref}}{\mapsto} sum\} : app\ y$
  - $cc_{ref}, \{\Gamma, y \overset{cc_{ref}}{\mapsto} sum\} : app$
    $\{\Gamma, y \overset{cc_{ref}}{\mapsto} sum\} : \lambda f_1.f_1\ l, cc_{app}$
  - $cc_{app}, \{\Gamma, y \overset{cc_{ref}}{\mapsto} sum\} : y\ l$
    - $cc_{app}, \{\Gamma, y \overset{cc_{ref}}{\mapsto} sum\} : y$
      - $cc_{ref}, \{\Gamma\} : sum$
        $\{\Gamma\} : \lambda xs_1.\widehat{e}_{sum}[xs_1/xs], cc_{ref}$
      $\{\Gamma, y \overset{cc_{ref}}{\mapsto} \lambda xs_1.\widehat{e}_{sum}[xs_1/xs]\} : \lambda xs_2.\widehat{\widehat{e}}_{sum}[xs_2/xs], cc_{ref}$
    - $cc_{ref}, \{\Gamma, y \overset{cc_{ref}}{\mapsto} \lambda xs_1.\widehat{e}_{sum}[xs_1/xs]\} : \widehat{\widehat{e}}_{sum}[l/xs]$
      $\vdots$

The `let`-binding of the argument *sum* ensures that the value substituted in the body of *app* refers to a heap binding annotated with the referring cost centre cc_{ref} attached. This is returned with the (renamed) binding of *sum* when y is applied.

Unfortunately the transformation has introduced some extra evaluation costs that would not be incurred during normal execution. Though we can discount any fixed costs introduced by the transformation (such as the heap allocation) there is some distortion of the execution time. We can omit the transformation if we can determine at compile that the actual application site will have the same cost centre as the reference site. However, there is still a (small) price to pay for correct cost attribution.

This problem does not arise with evaluation scoping since the cost centre returned with a function value is not attributed with the cost of evaluating the body of the function. Evaluation scoping always attributes these costs to the application site.

4.3 Lexical vs. Evaluation Scoping

The two profiling schemes attribute different costs to an annotated source expression. This difference can be summarised as follows:

> **Lexical scoping** attributes costs to the scope enclosing the *declaration site.*
>
> **Evaluation scoping** attributes costs to the scope enclosing the *application site.*

	Source Expression	Cost attributed to *cc*	
		Evaluation Scoping	Lexical Scoping
a)	scc *cc* $(\lambda x.e)$ y	$A_{\lambda x.e}$	$A_{\lambda x.e}$
b)	(scc *cc* $\lambda x.e$) y	0	$A_{\lambda x.e}$
c)	$(\lambda x.$scc *cc* $e)$ y	E_e	E_e
d)	scc *cc sum* l	A_{sum}	A_{sum}
e)	(scc *cc sum*) l	0	A_{sum}
f)	scc *cc* $(val, \lambda x.e)$	$E_{()} + E_{val} + 0$	$E_{()} + E_{val} + A_{\lambda x.e}$

E_{expr} is the cost of *E*valuating *expr* to ANF.
A_{fn} is the cost of *A*pplying the function *fn*.

Figure 4.6: Lexical vs. Evaluation Cost Attribution

However, the practical implications of this distinction are not immediately obvious. This section attempts to provide some insight into this difference, comparing the two schemes and summarising the relative merits. Our preference is for lexical scoping since its cost attribution seems more suitable for profiling, and is easier to maintain across the compiler transformations.

4.3.1 Some examples

Figure 4.6 presents the costs attributed to the annotating cost centre, *cc*, for some example expressions using the different profiling schemes. Examples (a), (b) and (c) in Figure 4.6 highlight the basic differences between the schemes.

(a) If a function is declared and applied within the scope of the same cost centre both schemes attribute the cost of applying the function to that cost centre.

(b) The function $\lambda x.e$ is declared within the scope of *cc* but applied outside that scope. The cost of applying the function is only attributed to *cc* under lexical scoping. Evaluation scoping attributes it to the scope of the particular application.

(c) Explicitly annotating the body of a function attributes the evaluation of the body to the cost centre *cc* under both schemes, regardless of where the function is applied.

Examples (d) and (e) are equivalent to Examples (a) and (b) respectively. Since *sum* is a top-level function the evaluation of **sum** is subsumed as if it was unfolded at the reference site (Section 4.1.4). There is no equivalent to Example (c) since annotating the body of

the declaration of *sum* would capture the costs of *all* references to *sum*, not just those arising from this reference.

Finally, example (f) shows the cost attribution of a data structure containing a function. Both schemes attribute the cost of evaluating the data structure to ANF to *cc*, but only lexical scoping attributes the cost of applying the function embedded within the resulting data structure to the cost centre *cc*. Evaluation scoping attributes the cost of applying the function to the scope of the application site. We discuss this further in Section 4.4.2.

4.3.2 Identifiable costs

The two profiling schemes enable the programmer to identify different costs. Evaluation scoping provides a very fine breakdown of particular costs, while lexical scoping provides a very useful aggregation of all the costs associated with executing the code in the scope of the `scc` annotation.

Particular applications

Evaluation scoping can distinguish between the costs of different applications of a particular function. In the example:

```
apph l1 l2 = scc "leth" let h x = ...
                        in (scc "h1" h l1, scc "h2" h l2)
```

evaluation scoping attributes the costs of each application of `h` to the cost centres enclosing each application site.

In contrast, lexical scoping attributes the costs of both the applications of `h` to the declaring cost centre `"leth"`. The costs of the different applications can only be distinguished if different versions of the function `h` are declared with different cost centre annotations or if `h` is declared at the top-level and treated as a function whose costs are subsumed. These solutions require undesirable reformulation of the source program.

Evaluation of functions

Evaluation scoping can also distinguish between the cost of evaluating a function to whnf and the cost incurred in applying the function. Consider our original example:

```
mapg x l = scc "mapg" (map (g x) l)
```

If g is a top-level function that takes one argument, examines it and returns a specialised function based on that value then evaluation scoping can distinguish the costs of specialising the function from the cost of mapping it over the list.

```
mapg' x l = scc "mapgx" (map (scc "gx" g x) l)
```

If instead, g requires two arguments then "gx" only measures the small cost of building the partial application.

Under lexical scoping we cannot distinguish the costs of evaluating a function from the cost of applying it. The cost of specialising `g x` *and* all the applications of `g x` is attributed to the cost centre "gx". The only way to distinguish the cost of applying the specialised function would be to annotate the body of the specialised function at the declaration site within g with a different cost centre. Unless the code for g is duplicated, this would cause *all* applications of that specialised function to be attributed to that cost centre, not just those arising from *this* reference to g.

Total cost of an expression

In contrast to the fine breakdown of costs identifiable by evaluation scoping, lexical scoping identifies the "total cost" associated with executing the code in the scope of an scc expression. If the programmer wants to identify this "total cost" they just have to annotate the declaration without worrying about the application sites of any functions that might be returned in the result. For example, consider the following declaration:

```
mapf x l = scc "mapf" map f l
```

Now suppose the programmer wants to identify the cost of all the applications of the top-level function `f` being passed to `map`. Under lexical scoping all the programmer has to do is annotate the reference to `f` being passed to `map`:

```
mapf x l = scc "mapf" map (scc "f" f) l
```

This ensures that the cost of all the applications of `f` in `map` are attributed to the cost centre "f".

In contrast, the fine breakdown of costs provided by evaluation scoping, makes the task of identifying the "total cost" more difficult. If the programmer wants to measure the cost of applying a function the application site must be annotated. In the example above the annotation `scc "f" f` only measures the cost of evaluating the partial application of `f` to no arguments. The cost of applying `f` is still attributed to the `"mapf"` cost centre. The application site can only be annotated if a specialised version of `map` is created with the application site annotated or a λ-abstraction is introduced that exposes and annotates the application of `f` in the function passed to `map`:

```
mapf x l = scc "mapf" map (\y -> scc "f" f y) l
```

Both of these solutions require undesirable reformulation of the source program. They are also rather confusing to the programmer.

4.3.3 Higher order functions

The two schemes also differ in their treatment of higher order functional arguments.

- Lexical scoping attributes the cost of applying a functional argument to the scope that declared it (or the scope which referenced it in the case of top-level subsumed functions).
- Evaluation scoping attributes the cost of applying a functional argument to the scope where the function is applied (as it does for all functions). This application site might be far removed from the declaration site.

Consider the expression:

```
scc "renaming" let lookup str = ...
               in rename lookup code
```

Evaluation scoping attributes the costs of looking up strings with the `lookup` function to the application site deep within `rename`. Modifying the definition of `lookup` affects the costs attributed to the cost centre of this application site, not the cost centre `"renaming"` which encloses the declaration site.

Lexical scoping attributes all the `lookup` costs to the scope of the declaration site i.e. the cost centre `"renaming"`. Any modifications to the `lookup` function would be reflected by changes in the costs attributed to `"renaming"`.

4.3.4 Transformation

Another important consideration is how easy it is to maintain the required attribution of costs during the transformation phases of the compiler (Section 5.4). The fine breakdown of costs provided by evaluation scoping makes the cost attribution more difficult to maintain during transformation — more optimisations have to be curtailed (see Section 5.4.7). In contrast, the attribution semantics provided by lexical scoping are much easier to maintain during transformation.

4.3.5 Implementation

The implementation of the two profiling schemes requires the cost centres to be manipulated in different ways (Section 5.5). This requires different code generator and runtime system modifications (though there were a lot of modifications that were common to both schemes). Both implementations pose specific problems that have to be overcome, but neither pose any particularly nasty difficulties.

4.3.6 Conclusion

We believe that lexical profiling has a more appropriate cost attribution for profiling. Its identification of "total cost" is a much more intuitive cost semantics since it corresponds to our intuitions about the cost of executing the "code" of the expression. It is easier to use and imposes fewer transformation restrictions. (But see Section 4.4.)

We have not found a significant need for the more detailed breakdown of costs provided by evaluation scoping. In fact, this detailed breakdown tends to hinder the process of cost centre annotation as the programmer has to be very careful that the annotations identify the costs they actually intend to measure.

4.4 Problems with Lexical Cost Attribution

Unfortunately lexical profiling still has its problems, that turn out to be quite significant in practice.

The evaluation of an expression instance can be broken down into two components:

- The one-off evaluation to ANF, and
- The repeated application of any functions embedded in the result.

Both profiling schemes attribute the one-off evaluation to ANF to the enclosing cost centre. However, the profiling schemes differ in their attribution of the costs associated with the application of any functions embedded in the result. Evaluation scoping attributes these costs to the cost centre of the application site, while lexical scoping continues to attribute them to the declaring cost centre.

Problems with the lexical attribution of embedded functions are discussed below. A hybrid solution that uses evaluation attribution at the particular places where it is deemed more suitable is then proposed in Section 4.4.3.

4.4.1 CAFs

The automatic annotation of CAFs (Section 4.1.7) is simple and effective. However, under lexical scoping the costs of functions embedded in the result of a CAF are attributed to the CAF cost centre. This turns out to be a very significant problem in practice. For example, the profile presented in Figure 6.1 attributes 78% of the execution time to the "`CAF:unicl`" cost centre (see Section 6.2.1).

Consider the alternative definitions of the function `unicl` used in the program `clausify` (see Section 6.1):

```
unicl1 formulae = filterset (not . tautclause)
                             (map clause formulae)

unicl2 formulae = (filterset (not . tautclause)
                  . map clause) formulae

unicl3 = filterset (not . tautclause) . map clause
```

The first two definitions declare `unicl` as a subsumed function. All the costs of applying

unicl are subsumed by the reference site. The third definition $\texttt{unicl}_3$ is a very simple CAF. It is annotated with a CAF cost centre:

```
unicl3 = scc_caf "CAF:unicl"
         filterset (not . tautclause) . map clause
```

Both profiling schemes attribute the costs of evaluating the compositions to the cost centre "CAF:unicl". But where will the costs of applying the resulting function be attributed? The answer depends on the profiling scheme being used.

- Evaluation scoping attributes the application of unicl to the site of its full application i.e. the current cost centre at the site where unicl is applied.
- Under lexical scoping the costs of applying unicl are attributed to the cost centre "CAF:unicl" because the code is referenced in the scope of the cost centre "CAF:unicl".

The cost attribution provided by lexical scoping is undesirable since the attribution differs significantly between the two, quite reasonable, definitions of unicl. Changing the definition of unicl from $\texttt{unicl}_2$ to $\texttt{unicl}_3$ (a simple η-reduction) would suddenly cause *all* the costs associated with unicl to be attributed to "CAF:unicl", rather than just the one-off costs of evaluating the composition application and updating the CAF. Moreover, this may be done automatically by the compiler optimisations since $\texttt{unicl}_3$ is more efficient because it only evaluates the function composition once.

This problem is not a result of introducing CAF cost centres. Indeed, removing the CAF cost centres only makes the problem worse since all the costs associated with the CAF, including the application of any functions embedded in the result, would then be attributed to the cost centre of the expression that happened to demand the value of the CAF first.

The unicl example might seem a little contrived, but there are situations where some significant one-off evaluation needs to be done before a partial application is returned. For example, a function that looks up builtin names might be expressed as:

```
lookup_Builtin :: String -> Identifier
lookup_Builtin = let hash_tbl = mk_Hash builtin_names
                 in lookup_Hash hash_tbl

mk_Hash :: Hash a => [(a, b)] -> (HashTbl a b)
mk_Hash mapping = ...

lookup_Hash :: Hash a => (HashTbl a b) -> a -> b
lookup_Hash tbl key = ...
```

The one-off construction of the hash table should be attributed to a CAF cost centre, but the repeated lookup costs should be subsumed by the application site.

4.4.2 Overloading

Haskell has a systematic way of handling overloading through the use of type classes (Wadler & Blott [1989]). Our experience with lexical scoping has identified a significant problem with the attribution of method costs.

A type class is a set of types sharing some operations, called methods. A `class` declaration specifies what the common operations are. A type is declared to be in the class with an `instance` declaration. The instance declaration describes what the methods in the class do for that particular type.

Figure 4.7 contains the declaration for the class `Eq` containing methods `(==)` and `(/=)`. Instance declarations are given for the types `Int` and `List a`. Note that the `List` instance requires an element type that is a member of the class `Eq`. The example also contains the definition for the overloaded function `elem` that determines if a value is an element of a list using the overloaded function `(==)`. A non-overloaded version `fnElem` that explicitly passes the equality function, is also given.

Dictionaries

The standard mechanism for implementing overloading has been to use method dictionaries (Hall et al. [1994]; Wadler & Blott [1989]), though various optimisations and alternative schemes have been proposed (Augustsson [1993]; Jones [1992]). Each overloaded function is given an extra argument that contains the methods for the particular type at which the function is being applied. The dictionary is given as an argument to the method, which

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  (/=) x y = if x == y then False else True

instance Eq Int where
  (==) x y = eqInt x y
  (/=) x y = neInt x y

instance (Eq a) => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  []     == ys     = False
  xs     == []     = False

elem :: Eq a => a -> [a] -> Bool
elem v l = scc "elem" case l of
             []   -> False
             x:xs -> (==) v x || elem v xs

fnElem :: (a -> a -> Bool) -> a -> [a] -> Bool
fnElem eq v l = scc "fnElem" case l of
                  []   -> False
                  x:xs -> eq v x || fnElem eq v xs
```

Figure 4.7: Example class, instance and use

```
(==) (m, _) = m
(\=) (_, n) = n
Eq.(==) dictEq x y = error "no =="
Eq.(/=) dictEq x y = if (==) dictEq x y then False else True

Eq.Int.(==) x y = eqInt x y
Eq.Int.(/=) x y = neInt x y
Eq.Int = (Eq.Int.(==), Eq.Int.(/=))

Eq.List.(==) dictEq []     []     = True
Eq.List.(==) dictEq (x:xs) (y:ys) =
    (==) dictEq x y && Eq.List.(==) dictEq xs ys
Eq.List.(==) dictEq []     ys     = False
Eq.List.(==) dictEq xs     []     = False

Eq.List.(/=) dictEq xs     ys     =
    if Eq.List.(==) dictEq xs ys then False else True

Eq.List dictEq = (Eq.List.(==) dictEq, Eq.List.(/=) dictEq)

elem dictEq v l = scc "elem" case l of
                    []   -> False
                    x:xs -> (==) dictEq v x || elem dictEq v xs

fnElem eq v l = scc "fnElem" case l of
                  []   -> False
                  x:xs -> eq v x || fnElem eq v xs
```

Figure 4.8: Translated class, instance and use

extracts the particular method from the dictionary. This is then applied to the method arguments as before.

Figure 4.8 gives a possible translation of the declarations in Figure 4.7. This defines: `(==)` and `(/=)` as selector functions; default `Eq` methods `Eq.(==)` and `Eq.(/=)`[6]; the `Eq.Int` and `Eq.List` methods and dictionaries; and adds a dictionary argument to the `elem` function. Note that the `List` methods and dictionary have an extra argument — the `Eq` dictionary for the element type.

Cost attribution

Let us consider the attribution of the method and dictionary costs in the application:

```
elem 10 intlist
```

This is translated to pass the appropriate element dictionary:

```
elem Eq.Int 10 intlist
```

Since `Eq.Int` is declared as a CAF (Figure 4.8), its declaration is annotated with a CAF cost centre, and the one-off construction costs attributed to this cost centre. Unfortunately lexical scoping also attributes the costs of applying the method functions to the CAF cost centre as they are referenced within the CAF. However, the programmer expects the application of the `(==)` method in the original source to be a top-level function with the costs being subsumed by the reference site within `elem` and attributed to the `"elem"` cost centre. The desired cost attribution is provided by evaluation scoping which attributes the costs of applying the method to the actual application site within `elem`.

In contrast, consider an application of `fnElem` where the equality function, `eqInt`, is passed explicitly in the original source:

```
fnElem eqInt 10 intlist
```

The costs of `eqInt` are attributed to the reference site (the cost centre enclosing the expression above) by lexical profiling or the application site (cost centre `"fnElem"`) by evaluation profiling. In this case the lexical attribution seems most appropriate since the costs are attributed to the cost centre enclosing the reference site.

[6] We disregard the restrictions Haskell places on identifier names.

Dictionary cost centres

The problem is worse when dictionaries cannot be declared statically. For example, searching for a singleton list in a list-of-lists might be defined as:

```
elem_ll :: Eq a -> a -> [[a]] -> Bool
elem_ll v ll = scc "elem_ll" elem [v] ll
```

This is translated to:

```
elem_ll dictEq v ll = scc "elem_ll" elem (EqList dictEq) [v] ll
```

A dictionary for comparing lists of the element type is built and passed to the `elem` function. The costs of constructing the dictionary is attributed to the cost centre `"elem_ll"`. However, the programmer is not necessarily aware of any dictionary construction costs attributed to the cost centre `"elem_ll"` since these were introduced by the compiler's implementation of overloading. Lexical scoping also attributes the costs of applying the `List.Eq.(==)` method to the `"elem_ll"` cost centre since the reference to `List.Eq.(==)` is embedded in the dictionary.

We solve the attribution of dictionary construction costs by attributing all these costs to special DICT cost centres. As for the CAF cost centres, a single `"DICT"` cost centre for each module is the default, but a compiler option is provided that annotates each dictionary construction with a cost centre derived from the name of the dictionary being built. Dictionaries declared as CAFs are annotated with a DICT cost centre, rather than a CAF cost centre. For example:

```
Eq.Int = scc_dict "DICT:Eq.Int" (Eq.Int.(==), Eq.Int.(/=))

elem_ll dictEq e ll = scc "elem_ll"
         elem (scc_dict "DICT:Eq.List" Eq.List dictEq) [e] ll
```

The $\texttt{scc}_{dict}$ annotation is similar to the $\texttt{scc}_{caf}$ annotation (Section 4.1.7) — it does not incrementing the sub-`scc` count of the enclosing cost centre.

With this dictionary annotation lexical scoping attributes *all* method application costs to the dictionary cost centre(s) responsible for building the dictionaries. Evaluation scoping still attributes the costs of applying methods to the application site.

4.4.3 A hybrid solution

The previous sections highlighted particular situations where evaluation scoping provided a more suitable cost attribution than lexical scoping. We now propose a hybrid profiling scheme based on lexical scoping but which uses evaluation scoping in exactly these circumstances. This provides us with the best of both worlds.

The idea is to attribute the costs of evaluating the body of any functions declared (or referenced) within the scope of a CAF or DICT cost centre to the cost centre of the application site rather than the declaring CAF or DICT cost centre. The costs of applying functions declared in the scope of any other cost centre are still attributed to the declaring cost centre (as required by lexical scoping).

The difference between lexical and evaluation scoping arose in the *Application* rule and this is precisely where we introduce the hybid profiling scheme. The hybrid *Application* rule is:

$$\frac{cc, \Gamma : e \Downarrow_{\theta_1} \Delta : \lambda y.e', cc_\lambda \qquad \text{EVAL}(cc_\lambda, cc), \Delta : e'[x/y] \Downarrow_{\theta_2} \Theta : z, cc_z}{cc, \Gamma : e\ x \Downarrow_{\{cc \mapsto \text{A}\} \uplus \theta_1 \uplus \theta_2} \Theta : z, cc_z}\ App_{hybrid}$$

$$\textbf{where} \quad \begin{array}{l} \text{EVAL}(\texttt{"CAF"}, cc) = cc \\ \text{EVAL}(\texttt{"DICT"}, cc) = cc \\ \text{EVAL}(cc_\lambda, cc) = cc_\lambda \end{array}$$

This has a runtime choice which chooses the cost centre to which the evaluation of the body of the λ-abstraction is attributed. This choice is based on the attribution properties associated with the cost centre, cc_λ, returned with the λ-abstraction — CAF and DICT cost centres use evaluation scoping while all the other cost centres use lexical scoping. This is captured by the EVAL selector.

Consider the evaluation of the method application `(==) dictEq v x` within the `elem` function using the App_{hybrid} rule above. The heap, Γ, will contain the following bindings:

$$\begin{array}{lll} \Gamma = \Delta, & & \\ \quad (==) & \overset{\texttt{"SUB"}}{\mapsto} & \lambda p.\texttt{case}\ p\ \texttt{of}\ (m, n)\ \texttt{->}\ m, \\ \quad dictEq & \overset{\texttt{"DICT"}}{\mapsto} & (Eq.Int.(==), Eq.Int.(/=)), \\ \quad Eq.Int.(==) & \overset{\texttt{"DICT"}}{\mapsto} & \lambda y.\lambda z.e_= \end{array}$$

We assume that the binding of the top-level method `Eq.Int.(==)` has been boxed and updated with a binding annotated with the `"DICT"` cost centre. The hybrid reduction sequence is:

$cc_{elem}, \{\Gamma\} : (==)\ dictEq\ v\ x$
| $cc_{elem}, \{\Gamma\} : (==)\ dictEq\ v$
| | $cc_{elem}, \{\Gamma\} : (==)\ dictEq$
| | | $cc_{elem}, \{\Gamma\} : (==)$
| | | $\{\Gamma\} : \lambda p_1.\texttt{case}\ p_1\ \texttt{of}\ (m_1, n_1)\ \texttt{->}\ m_1, cc_{elem}$
| | | $cc_{elem}, \{\Gamma\} : \texttt{case}\ dictEq\ \texttt{of}\ (m_1, n_1)\ \texttt{->}\ m_1$
| | | | $cc_{elem}, \{\Gamma\} : dictEq$
| | | | $\{\Gamma\} : (Eq.Int.(==), Eq.Int.(/=)), \texttt{"DICT"}$
| | | | $cc_{elem}, \{\Gamma\} : Eq.Int.(==)$
| | | | $\{\Gamma\} : \lambda y_1.\lambda z_1.\widehat{e}_{=}[y_1/y, z_1/z], \texttt{"DICT"}$
| | | $\{\Gamma\} : \lambda y_1.\lambda z_1.\widehat{e}_{=}[y_1/y, z_1/z], \texttt{"DICT"}$
| | $\{\Gamma\} : \lambda y_1.\lambda z_1.\widehat{e}_{=}[y_1/y, z_1/z], \texttt{"DICT"}$
| | $\boxed{cc_{elem}}, \{\Gamma\} : \lambda z_1.\widehat{e}_{=}[v/y, z_1/z]$
| | $\{\Gamma\} : \lambda z_1.\widehat{e}_{=}[v/y, z_1/z], cc_{elem}$
| $\{\Gamma\} : \lambda z_1.\widehat{e}_{=}[v/y, z_1/z], cc_{elem}$
| $cc_{elem}, \{\Gamma\} : \widehat{e}_{=}[v/y, x/z]$
| ⋮

The critical point in the reduction sequence occurs when the method extracted from the dictionary is applied to its first argument. At this point we observe that the function being applied was returned with a "DICT" cost centre and evaluate the function body in the context of the application cost centre cc_{elem} (observe the $\boxed{\phantom{cc_{elem}}}$ above).

Implementation

Apart from the introduction of a simple runtime test, the implementation of this hybrid profiling scheme poses no particular problems (see Sections 5.4.8 and 5.5.5).

4.5 Conclusion

This Chapter has developed an abstract semantics of cost attribution that is independent of the underlying implementation (though the actual costs that are reported depend on the underlying implementation). Initially we compared two possible semantics that satisfied the principles of cost attribution:

Lexical Scoping attributes the cost of evaluation to the cost centre enclosing the "code" being executed i.e. the cost centre enclosing the declaration site.

Evaluation Scoping attributes the cost of evaluation to the scope enclosing the application site.

Since the practical implications of this distinction were not immediately obvious we have implemented and compared the practical use of the two profiling semantics (Chapters 5 and 6). Our conclusion is that lexical scoping is more suitable for profiling. Its identification of "total cost" is a much more intuitive cost semantics since it corresponds to our intuitions about the cost of executing the "code" of the expression. It is easier to use and imposes fewer transformation restrictions.

However, practical experience also identified a significant problem with lexical profiling: the costs of applying functions embedded in the result of a CAF are attributed to the CAF's cost centre, rather than being subsumed by the reference site. This is not a problem for evaluation scoping because it attributes the costs of applying all functions, including those embedded in CAF results, to the application site. In response to this, we have developed a hybrid profiling scheme that is base on the lexical cost attribution, but uses evaluation cost attribution for those functions declared inside a CAF or DICT cost centre. This should significantly improve the practical usability of the profiler since this cost attribution corresponds more closely to the programmer's intuition.

It is worth noting that the formal cost semantics has proved invaluable in providing insight, and enabling a precise formulation of the rather subtle distinction between evaluation scoping and lexical scoping. The benefit of using the formal semantics was clearly demonstrated by the almost trivial ease with which the hybrid profiling scheme was subsequently developed.

Chapter 5

Implementation

The implementation of the profiler was a significant undertaking, requiring modifications to both the compiler and runtime system.

- The source expressions to be profiled are identified early in the compilation process. (Section 5.2)
- The attribution of the costs of these expressions is preserved by the transformations performed during compilation. (Sections 5.3 and 5.4)
- Code is generated that identifies and maintains the *current cost centre* during execution. (Section 5.5)
- The runtime system is extended to gather the profiling data. (Section 5.6)

The documentation for the various profiling related compiler options and runtime system options is presented in Appendix B.

Before describing the implementation of the profiler we give a brief overview of the structure of the Glasgow Haskell compiler to provide a framework for describing the modifications required for profiling.

5.1 The Glasgow Haskell Compiler

The Glasgow Haskell compiler is a state-of-the-art optimising compiler. It has been developed at the University of Glasgow as part of the GRASP project funded by the Science and Engineering Research Council (SERC). The major characteristics of the compiler are:

- The compiler is written almost entirely in Haskell. The only exception is that the parser is written in Yacc and C. Its core, `hsc`, consists of a number of distinct passes (see Figure 5.1), each responsible for a different aspect of the compilation. Within the compiler monads are used extensively to carry around the "plumbing" and catch any compilation errors (Wadler [1990]). The reader is referred to Peyton Jones et al. [1993] for a description of these passes and an overview of the main compilation techniques used.

- The code generated is based on the Spineless Tagless G-machine model of reduction (Peyton Jones [1992]). The operational semantics of the STG-machine are presented in Appendix A.2.

- The compiler generates C as its target code, providing a high level of portability. However, the very significant cost of compiling the C code have prompted the development of simple native assembly code generators for common architectures.

- The runtime system is also written in C. It includes a highly configurable interface between the storage manager and the compiler that comes with a number of different garbage collectors, including a generational collector (Sansom & Peyton Jones [1993]).

- Other features include:

  - Mixed language programming, with a C interface.
  - Monolithic and incrementally-updateable arrays with $O(1)$ access time.

The overall organisation of the compiler is quite conventional. A driver program processes the compiler options and runs a sequence of Unix processors, namely: a "literate-script" pre-processor, the Lex/Yacc parser, the core of the compiler (`hsc`), the C compiler and Unix assembler, and the Unix linker (see Figure 5.1).

The profiler was developed as an integral part of the compiler. Since the profiled expressions are identified in the original source, modifications were required throughout the compiler because every pass had to be extended to deal with the `scc` expression construct. The most significant modifications were in the transformation passes and the

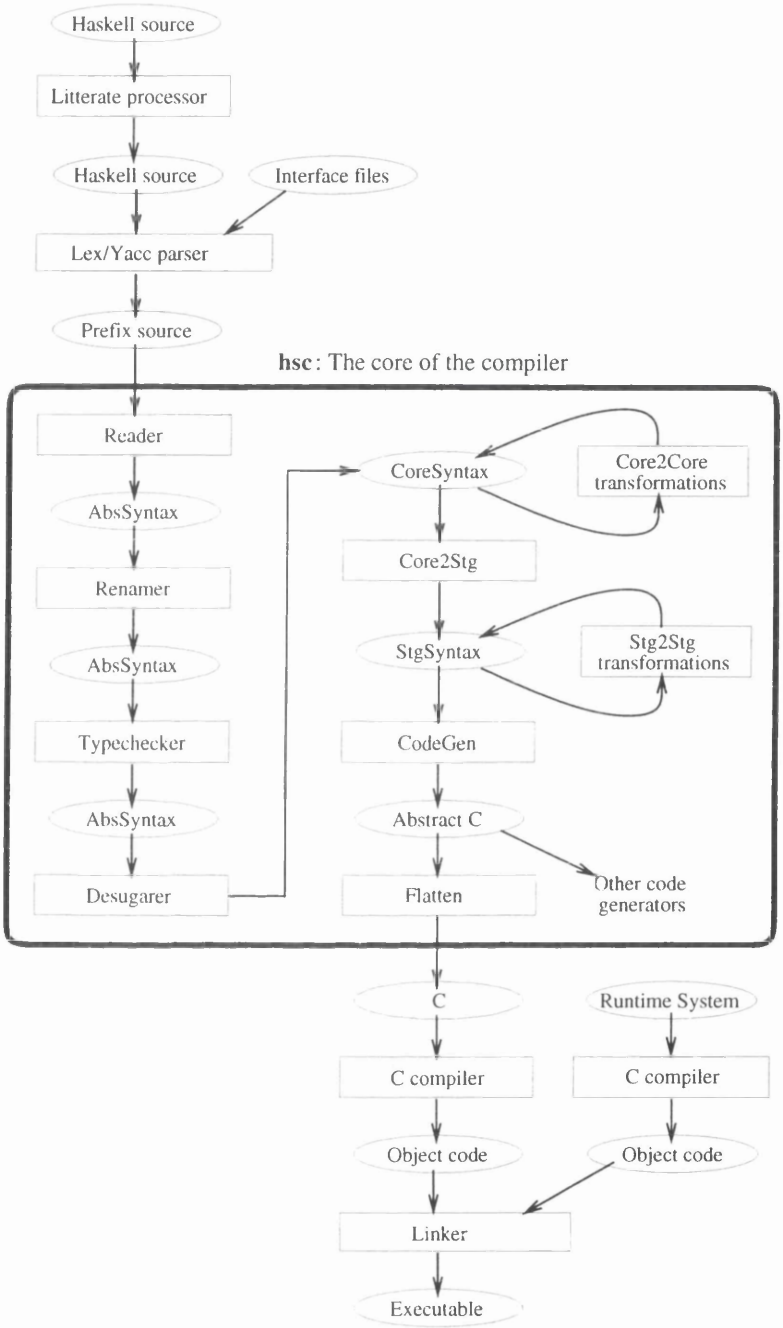


Figure 5.1: Structure of the Glasgow Haskell compiler

code generator. In addition the runtime system was extended to gather and report the profiling data during execution.

5.2 Identifying Source Expressions

In this section we describe how the programmer identifies the particular source expressions of interest. The mechanisms provided to the programmer for identifying expressions are independent of the underlying notion of `scc` annotations (described in Chapter 4), since the front end of the compiler can easily introduce the appropriate `scc` annotations.

The profiling tool currently provides two methods for identifying the expressions to be associated with cost centres.

5.2.1 Automatic annotation

A compiler option can be used to instruct the compiler to annotate the body of each top-level function definition with a cost centre of the same name (see Appendix B.1). Recording the module name with each cost centre enables the cost of the module as a whole to be determined by summing the costs of the individual cost centres in the module.

5.2.2 Explicit `scc` annotations

Alternatively the programmer can explicitly annotate expressions in their source code using the `scc` expression construct directly. This enables the programmer to annotate any expression of interest. If the program has a clear logical structure, such as the passes of a compiler, a few `scc` annotations at the top level can quickly reveal which "parts" of the program should be focussed on.

5.2.3 Expressions vs. Functions

We do not believe that the decision to allow the user to annotate expressions, rather than named functions, is particularly significant — it only affects the profiling interface. We chose to allow the programmer to annotate expressions directly because Haskell is an expression oriented language. The ability to identify expressions reduces the need to massage the code to identify expression of interest. For example, annotation of a particular branch of a `case` is very straight forward with this expression oriented approach. Of course,

the expressions identified may themselves be functions. Indeed, the automatic annotation compiler option identifies and annotates all top-level functions.

Within the compiler the expression-oriented view is more significant. It is important that a cost centre can be associated with an expression, or some sub-expression, as it is manipulated (see Section 5.3). It has proved very useful to make this identification explicit, as this gives us a language in which we can express the manipulation of cost centres within the compiler.

5.2.4 Possible extensions

There are many other possible methods of identifying expressions. Other possibilities we have considered include:

- Source annotations that direct the compiler to annotate each binding in a `let` or `where` construct with a cost centre of the name being bound.
- Compiler options that name the functions to be annotated or point to a file containing the names to be annotated.
- Cost centres that can be activated/deactivated at runtime. A number of profiles with different costs being identified could then be generated without requiring recompilation.

None of these extensions is particularly difficult to implement and they may well improve the usability of the profiling tool quite considerably. However, we have deferred any further implementation until a real need for a particular extension is identified.

5.3 Transformation and Optimisation

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

- Many transformations do not change the attribution of costs and can proceed as normal.

- We can still perform arbitrary program transformations on the expression within an `scc` annotation and on expressions containing `scc` annotated sub-expressions. The restrictions only affects transformations occurring across the cost centre boundaries.

- It is possible to relax the transformation restriction provided any sub-expressions that are moved into the scope of a different cost centre are annotated with the cost centre of their original scope. Thus, many transformations can still be performed even when they do move evaluation across a cost centre boundary.

A key advantage of our approach is that program transformations are only hindered by the actual `scc` annotations introduced by the programmer (either by explicit source annotation or implicitly with a compiler flag). Thus, the program being profiled may differ from a fully optimised version, but only at the `scc` boundaries. The optimisation of a large program containing a few careful `scc` annotations proceeds largely unhindered. Unfortunately care must still be taken since one lost transformation might stall a whole series of subsequent optimisations, possibly in the "inner loop" of the program.

5.3.1 Cost centre boundaries

The `scc` expression construct identifies the cost centre boundaries. The very explicit nature of the construct has a number of advantages:

- It necessarily requires us to examine the treatment of a cost centre boundary in *every* pass in the compiler!

- It provides us with a language that can be used to express alternative versions of transformations that would move evaluation from the scope of one cost centre to another. These modified transformations must maintain the appropriate cost attribution.

For example, consider the following transformation:

$$\texttt{scc}\ cc\ ...E_{sub}... \implies \texttt{let}\ v = E_{sub}\ \texttt{in}\ \texttt{scc}\ cc\ ...v...$$

Though this transformation doesn't change the meaning of the program, but it does change the attribution of evaluation costs. The costs of evaluating E_{sub} are no longer attributed to cc since E_{sub} has been lifted outside the `scc` annotation. However, it is still possible to

perform this transformation if the expression E_{sub} is annotated with the cost centre of its original scope:

$$\texttt{scc}\ cc\ ...E_{sub}... \implies \texttt{let}\ v = \texttt{scc}_{sub}\ cc\ E_{sub}\ \texttt{in}\ \texttt{scc}\ cc\ ...v...$$

This ensures that the costs of evaluating E_{sub} are still attributed to the cost centre cc.

5.3.2 Annotating sub-expressions

In general, a sub-expression can be moved into the scope of a different cost centre if it is annotated with the cost centre enclosing its original scope. An $\texttt{scc}_{sub}$ annotation is used to annotate these moved sub-expressions. The effect of an $\texttt{scc}_{sub}$ annotation is only to attribute the evaluation costs to the appropriate cost centre. Evaluating an $\texttt{scc}_{sub}$ does not increment the count of the expression instances evaluated — this is only incremented when the original scc expression is entered (see Section 4.1.2). Since they do not keep track of entry counts $\texttt{scc}_{sub}$ annotations can be eliminated if there is no cost involved in evaluating the expression they are annotating. They can also be eliminated if they reside within the scope of an scc with the same cost centre. This may arise if subsequent transformations move the sub-expression back into the scope of the original cost centre.

There are two distinct changes of scope:

- A sub-expression that is lifted out of an scc expression. The sub-expression can always be annotated with the cost centre of the scc expression (see above).
- An expression that is unfolded or inlined inside an scc sub-expression:

$$\texttt{let}\ v = E_{sub}\ \texttt{in}\ \texttt{scc}\ cc\ ...v... \implies \texttt{scc}\ cc\ ...(\texttt{scc}_{sub}\ ecc\ E_{sub})...$$

Unfortunately the enclosing cost centre, ecc, may not always be known at compile time. If the expression that we wish to unfold does not reside within an scc annotated expression the costs of evaluating the sub-expression are subsumed by the cost centre which referenced the function. If this cannot be determined at compile time the unfolding cannot be performed by the compiler (but see Section 5.4.4).

Though this may seem restrictive, there are some important special cases that can always be unfolded. Since the costs of top-level functions are always subsumed by the reference site (see Section 4.1.4) they can always be unfolded at the reference site. Thus,

the inlining of top-level function declarations is not hindered! In addition, simple bindings that do not involve any evaluation or heap allocation, such as variable renamings, (unboxed) literal values, and zero-arity constructors, can always be inlined.

5.4 Transformation in the Glasgow Haskell compiler

The compiler first translates a Haskell program into a Core Language (Figure 5.2), removing all syntactic sugar. The Core Language is a variant of the second order lambda calculus augmented with the constructs `let(rec)`, `case` and `scc`. It is designed to aid program transformation by making certain information explicit:

- All application arguments are atomic. This forces the creation of argument closures to be made explicit using `let` bindings.
- The boxing and unboxing of values is made explicit. This enables many low-level transformations usually relegated to the code generator to be expressed as Core-to-Core transformations.

Most of the optimising program transformations within the compiler are performed on the Core Language (Figure 5.2). At the heart of the compiler is a set of local transformations that simplify core expressions. In addition to these there are some more specialised transformations aimed at particular optimisations:

- Let bindings may be floated outward to increase sharing or inwards to avoid unnecessary allocation.
- The worker/wrapper transformation arranges for strict function arguments to be passed unboxed.
- Intermediate list data structures are eliminated using foldr/build deforestation.

As already noted the explicit `scc` construct requires us to add code to deal with an `scc` annotation in *every* pass within the compiler. This makes it quite difficult to overlook the introduction of `scc` annotations and perform "bad" transformations which modify the attribution of costs. The following sections discuss the transformations focussing on the preservation of the cost attribution for lexical profiling. The preservation of evaluation

Program	$prog \rightarrow binds$	
Bindings	$binds \rightarrow bind_1 ; \ldots ; bind_{n \geq 1}$	
	$bind \rightarrow var$ `=` $expr$	
Expression	$expr \rightarrow$ `let` $bind$ `in` $expr$	Local definition
	| `letrec` $binds$ `in` $expr$	Local recursion
	| `\` $var_1 \ldots var_{n \geq 1}$ `->` $expr$	Lambda abstraction
	| $expr\ atom$	Application
	| $\Lambda\ tyvar$ `->` $expr$	Type abstraction
	| $expr\ ty$	Type application
	| `case` $expr$ `of` $alts$	Case expression
	| $constr\ atom_1 \ldots atom_{n \geq 0}$	Saturated constructor
	| $prim\ atom_1 \ldots atom_{n \geq 0}$	Saturated built-in op
	| `scc` $cc\ expr$	Set cost centre
	| $literal$`#`	Unboxed object
	| var	
Alternatives	$alts \rightarrow aalt_1 ; \ldots ; aalt_{n \geq 0} ; [def]$	Algebraic alts
	| $palt_1 ; \ldots ; palt_{n \geq 0} ; [def]$	Primitive alts
Algebraic alt	$aalt \rightarrow constr\ var_1 \ldots var_{n \geq 0}$ `->` $expr$	
Primitive alt	$palt \rightarrow literal$`#` `->` $expr$	
Default alt	$def \rightarrow var$ `->` $expr$	
Atom	$atom \rightarrow literal$`#` | var	*Atomicarguments*

Figure 5.2: Syntax of the Core language

profiling cost attribution and our hybrid cost attribution are considered separately in Sections 5.4.7 and 5.4.8 respectively.

5.4.1 Local transformations

The "simplifier" consists of a set of very simple Core-to-Core transformations (see Figure 5.3). It proceeds in two phases:

- The program is analysed to determine the way each named value is used. This includes both occurrence counts and basic strictness information. This information is used to ensure that the constraints on particular transformations are satisfied and to identify the `let`-bindings to be inlined. Bindings are normally inlined if they occur once or they are bound to a variable, literal, or zero-arity constructor. A binding that does not occur at all is removed.

  More aggressive unfolding heuristics may be used when unfolding lambda abstractions at the expense of possible code duplication (Santos & Peyton Jones [1992]). In particular, top-level function definitions may be inlined if it is expected that this will lead to further optimisation.

- Based on this information the program is then simplified, using the set of transformations in Figure 5.3.

Since one transformation pass may expose further transformations, this process is iterated until no more transformations are applicable or a user-specifiable maximum number of iterations (default 4) have been performed.

Transformation with `scc`

The transformation restriction states that: *evaluation must not be moved from the scope of one cost centre to another.* Many of the local transformations can be applied without modification as their effect is limited to the transformation of local language constructs and they do not modify the sub-expressions within. There are two situations where cost centres and `scc` annotations affect these local transformations:

- Unfolding and case elimination perform a substitution on the entire expression. This may move evaluation into the scope of another `scc` annotation. β-reduction also

Name	Before	After
Unfolding or Inlining[1]	`let` v = E_v `in` E	$E\ [E_v/v]$
Case Elimination[2]	`case` E_v `of` v -> E	$E\ [E_v/v]$
β-reduction	$(\lambda v.E)\ x$	$E\ [x/v]$
Let to Unboxing Case[3]	`let` v = E_v `in` E	`case` E_v `of` $C\ v_1 \ldots v_n$ -> `let` v = $C\ v_1 \ldots v_n$ `in` E
Let to Case[4]	`let` v = E_v `in` E	`case` E_v `of` v -> E
Constructor Reuse I	`let` v = $C\ v_1 \ldots v_n$ `in` ... $C\ v_1 \ldots v_n$ ...	`let` v = $C\ v_1 \ldots v_n$ `in` ... v ...
Constructor Reuse II	`case` v `of` ... $C\ v_1 \ldots v_n$ -> ... $C\ v_1 \ldots v_n$ ...	`case` v `of` ... $C\ v_1 \ldots v_n$ -> ... v ... ...
Case of known constructor	`case` $C_i\ v_1 \ldots v_n$ `of` ... $C_i\ v_{i1} \ldots v_{in}$ -> E_i ...	$E_i\ [v_1/v_{i1} \ldots v_{in}/v_n]$
Let Floating from Let	`let` v = `let` w = E_w `in` E_v `in` E	`let` w = E_w `in let` v = E_v `in` E
Let Floating from Case	`case (let` v = E_v `in` E`) of` ...	`let` v = E_v `in case` E `of` ...
Let Floating from App	`(let` v = E_v `in` E`)` x	`let` v = E_v `in` $E\ x$
Case Floating from Let[5]	`let` v = `case` E_c `of` alt_1 -> E_1 ... alt_n -> E_n `in` E	`case` E_c `of` alt_1 -> `let` v = E_1 `in` E ... alt_n -> `let` v = E_n `in` E
Case Floating from Case (Case of Case)	`case` (`case` E_c `of` alt_{c1} -> E_{c1} ... alt_{cm} -> E_{cm}) `of` alt_1 -> E_1 ... alt_n -> E_n	`case` E_c `of` alt_{c1} -> `case` E_{c1} `of` alt_1 -> E_1 ... alt_n -> E_n ... alt_{cm} -> `case` E_{cm} `of` alt_1 -> E_1 ... alt_n -> E_n
Case Floating from App	(`case` E_c `of` alt_1 -> E_1 ... alt_n -> E_n) v	`case` E_c `of` alt_1 -> $E_1\ v$ ... alt_n -> $E_n\ v$

[1] See Section 5.4.1.

[2] v used strictly in E and either v occurs only once in E or E_v is a constant or variable.

[3] v used strictly in E and has a type with a single constructor, C.

[4] v used strictly in E.

[5] v used strictly in E or E_c is a "cheap" primitive operation that cannot fail. If v is recursive then v must not occur in E_c.

Figure 5.3: Local Transformations

performs a substitution on the resulting expression but it is only substituting one variable for another which does not change the attribution of costs.

- The applicability of transformations that match patterns consisting of more than one language construct may be hindered by an intervening `scc` construct.

Substitution in sub-expressions

The unfolding and case elimination transformations perform a substitution on the result expression.

$$\texttt{let } v \texttt{ = } E_v \texttt{ in } E \implies E\,[E_v/v]$$
$$\texttt{case } E_v \texttt{ of } v \texttt{ -> } E \implies E\,[E_v/v]$$

If the expression being substituted E_v is not a simple variable, literal, or zero-arity constructor and the variable v occurs in the scope of a different cost centre, E_v *must* be annotated with the enclosing cost centre of the `let` or `case` if it is to be substituted. If the enclosing cost centre is not known at compile time substitution cannot proceed and the transformation must not be applied.

This restriction is enforced by extending the simplifier's analysis phase to determine which bindings can be safely substituted before performing the transformation. The occurrence count information for each binding is split into two counts:

- this-`scc`: occurrences within the scope of the enclosing cost centre and
- sub-`scc`: occurrences in the scope of `scc` annotated sub-expressions.

This information is then used to determine when the substitution is safe and the transformation can be applied. If the variable occurs in the scope of a sub-`scc` expression the substituted expression E_v is annotated with the enclosing cost centre *ecc*. If the enclosing cost centre is not known the substitution cannot be performed. This is summarised in Figure 5.4.

Intervening `scc`s

If an `scc` annotation interferes with the pattern of constructs required by a transformation the transformation is not applied since the constructs don't match. This is important as

Occurrence Counts			Enclosing	Substitution
total	this-scc	sub-scc	Cost Centre	
0	0	0		E
1	1	0		$E\ [E_v/v]$
1	0	1	Known	$E\ [\mathtt{scc}_{sub}\ ecc\ E_v/v]$
1	0	1	Unknown	cancelled
>1	>1	0		n.a.[1]
>1		>1	Known	n.a.[2]
>1		>1	Unknown	n.a.[3]

[1]Lambda abstractions may still be inlined.
[2]More aggressive unfoldings require the expression substituted to be annotated with the enclosing cost centre.
[3]More aggressive unfoldings have to be cancelled.

Figure 5.4: Substituting with Cost Centres

it prevents "bad" transformations being performed. For example, in the expression

$$\texttt{case (scc } cc \texttt{ let } v = E_v \texttt{ in } E\texttt{) of } alts$$

the let-floating-from-case transformation is prevented by the presence of the scc annotation.

If we still want the local transformation to be performed, we have to introduce an additional transformation rule that matches the scc construct. In introducing this rule we are forced to consider the implications of the scc and the required attribution of costs in the resulting expression. An example scc transformation is discussed in Section 5.4.3.

5.4.2 Effect of scc on local transformation

The optimisations that are curtailed depend on the placement of cost centres. Figure 5.5 compares the number of local transformations performed when compiling `clausify` (described in Section 6.1) with different scc annotations. The first column reports the number of transformations with no scc annotations. This is exactly the same as the number of transformation performed during normal, unprofiled compilation. The Explicit scc column reports the number of transformations performed when five explicit scc annotations are added to the source (see Section 6.1) and the Automatic scc column reports the number of transformations performed when all top-level declarations are annotated. Examination of Figure 5.5 reveals that:

Transformation	No scc No.	Explicit scc No.	Change	Automatic scc No.	Change
Unfolding	424	408	−16	391	−33
Unused Binding	25	25		28	+3
Let to Case	0	0		0	
Constructor Reuse	4	4		4	
Case of Known Constr	16	16		14	−2
Let Float from Let	119	59	−60	29	−90
Let Float from Case	12	12		12	
Let Float from App	142	85	−57	64	−78
Case Float from Let	0	0		0	
Case Float from Case	18	18		14	−4
Case Float from App	0	0		0	
Execution Time (secs)	4.02	4.04	+0%	4.44	+10%

These figures are for ghc Version 0.16.

Figure 5.5: Effect of `scc` Annotations on Transformation of `clausify`

- Only 4% of the unfoldings are curtailed when the explicit `scc` annotations are introduced. Even with all top-level declarations annotated this figure only rises to 8%.
- The presence of `scc` annotations hinders a large number of `let` floating transformations. A solution to this problem is discussed in Section 5.4.3.
- A few `case` transformations are curtailed when all top-level declarations are annotated.

The final row in Figure 5.5 shows the profiled execution time. The increase in execution time with automatic annotation is due to the considerable bookkeeping required during the profiled execution rather than the curtailed optimisations (see Section 6.2.2).

5.4.3 Let floating

Figure 5.5 revealed that a significant number of local `let`-floating transformations were curtailed when the `scc` annotations were introduced. In addition to the local `let` floating transformations, there are also some global `let` floating transformations (Peyton Jones, Santos & Partain [1994]). These come in two flavours:

Floating outwards: Floating `let`-bindings out of λ-abstractions to improve sharing. This is similar to the full laziness transformation (Hughes [1983]; Peyton Jones & Lester [1990]).

Floating inwards: Floating `let`-bindings inwards to avoid allocating the binding unnecessarily.

To enable all these transformations to proceed without being hindered by `scc` annotations we introduce transformations that annotate the right-hand-side of a `let`-binding with an $\texttt{scc}_{sub}$ annoation when it is floated past an `scc` annotation:

$$\texttt{scc}\ cc\ \texttt{let}\ v = E_v\ \texttt{in}\ E \implies \texttt{let}\ v = (\texttt{scc}_{sub}\ cc\ E_v)\ \texttt{in}\ (\texttt{scc}\ cc\ E)$$

$$\texttt{let}\ v = E_v\ \texttt{in}\ (\texttt{scc}\ cc\ E) \implies \texttt{scc}\ cc\ \texttt{let}\ v = (\texttt{scc}_{sub}\ ecc\ E_v)\ \texttt{in}\ E$$

The second transformation, which floats a `let`-binding into an `scc` annotation, can only be performed if the enclosing cost centre, *ecc*, is known.

These let-floating `scc` transformations enable the other let-floating transformations to proceed unhindered. For example, the hindered let-floating-from-case transformation example given in Section 5.4.1 can now proceed as follows:

$\texttt{case}\ (\texttt{scc}\ cc\ \texttt{let}\ v = E_v\ \texttt{in}\ E)\ \texttt{of}\ alts$

$\implies$ let-floating-from-scc

$\texttt{case}\ (\texttt{let}\ v = \texttt{scc}_{sub}\ cc\ E_v\ \texttt{in}\ \texttt{scc}\ cc\ E)\ \texttt{of}\ alts$

$\implies$ let-floating-from-case

$\texttt{let}\ v = \texttt{scc}_{sub}\ cc\ E_v\ \texttt{in}\ \texttt{case}\ (\texttt{scc}\ cc\ E)\ \texttt{of}\ alts$

Though we annotate the body of the binding with an $\texttt{scc}_{sub}$ annotation, the allocation of the `let`-binding is moved into the scope of a different cost centre. This violates the principle of preserving the cost attribution. However, we believe that the benefit to program transformation is worth this movement in execution costs. We still attach the original cost centre to the closure, rather than the current cost centre, when the closure is allocated, and the space allocated is still attributed to the original cost centre. It is only the small amount of execution time required to allocate and initialise the closure that is attributed to the current cost centre.

Floating constant expressions (CAFs)

A constant expressions which is floated to the top-level is turned into a CAF. If it is floated out of an `scc` annotated declaration the transformations above ensures that it is annotated with its original cost centre. However, if it is floated from a subsumed scope, no $\texttt{scc}_{sub}$ annotation is attached. Instead the one-off evaluation costs are attributed to a CAF cost centre (Section 4.1.7).

Unfortunately the names of the cost centres of these introduced CAFs may be a source of confusion for the programmer. To alleviate this problem we plan to improve the naming of the bindings introduced by the compiler, basing them on the name of the enclosing declaration. This will result in the introduced CAF cost centres being given a more meaningful name.

5.4.4 Enclosing cost centres

The main obstacle to the specific `scc` transformations is not knowing the enclosing cost centre at compile time. This situation occurs in the scope of a top-level subsumed function when no explicit `scc` annotation encloses the expression being transformed. This is not a problem if the entire function body is annotated (as is the case with automatic annotation) since the enclosing cost centre is known. However, with explicit annotation there may be situations where an `scc` annotation occurs in a scope where the enclosing cost centre is not known.

One possible solution is to record the enclosing cost centre when the function is entered. Subsequent transformations can then annotate a sub-expressions with this recorded cost centre. This is easily expressed at the source level by allowing cost centres to be manipulated within the language. A `get_ccc` primitive is used to record the current cost centre enclosing a function when it is entered.

```
    f x = body
⟹ record enclosing cost centre
    f x = let ecc = get_ccc
          in body
```

If required, subsequent transformations can now annotate a sub-expression with the enclosing cost centre *ecc*: $\texttt{scc}_{sub}$ *ecc* e_{sub}. The problem with this approach is that the execution

may be distorted since *ecc* is now a free variable of any expression in which it is used and must be saved in any closures. Though the extra space allocation can be discounted in the profiling data reported, the execution time would be distorted. However, this may be a price worth paying if it enables additional program transformations to proceed unhindered.

5.4.5 Worker/Wrapper unboxing

The worker/wrapper transformation was developed to use strictness information to reduce the amount of boxing (Peyton Jones & Launchbury [1991]). The basic idea is to split each function definition into two pieces. The *wrapper* which takes normal boxed arguments, evaluates any strict arguments that have a single constructor, and passes the components to the worker. The *worker* takes the unboxed arguments and evaluates the body of the function, which has been optimised to use the unboxed values.

If the result has a single constructor of arity one it may also be returned in an unboxed form, and boxed by the wrapper. However, this is not implemented since the increased cost to our STG-machine implementation is negligible — the STG-machine returns these apparently boxed values in a register (Peyton Jones [1992]).

Consider the standard definition of factorial, with the boxing made explicit:

```
fac n = case n of
        Int n# -> case n# of
                  0#  -> Int 1#
                  n#' -> n * fac (n-(Int 1#))
```

Unboxed values are identified by the use of a trailing `#`. Boxed values are constructed using normal data constructors applied to unboxed values. For example, the boxed integer `1` is expressed as `Int 1#`.

The worker/wrapper transformation observes that `fac` is strict in `n` and splits this definition into:

```
fac n = case n of Int n# -> fac# n#

fac# n# = let n = Int n#
          in
          ...original body of fac...
```

The subsequent transformation of the worker, which includes the unfolding of the wrappers of `-`, `*` and the recursive call to `fac`, gives:

```
fac# n# = case n# of
          0#  -> Int 1#
          n#' -> case (n# -# 1#) of
                 n1# -> case (fac# n1#) of
                        Int m# -> case (n# *# m#) of
                                  r# -> Int r#
```

The transformations have removed the boxing of the arguments to `fac`, `-` and `*`. The result of `fac#` is still a boxed value, but the STG-machine returns this in a register — never constructing it in the heap.

Transformation of scc annotated definitions

How does the introduction of cost centre annotations affect this optimisation? Let us consider a definition of `fac` with the entire body annotated with a cost centre.

```
fac n = scc "fac" case n of
                  Int n# -> case n# of
                            0#  -> Int 1#
                            n#' -> n * fac (n-(Int 1#))
```

The splitting process is modified to identify the `scc` annotation of the body and annotate the wrapper with the `scc`. The worker is not annotated with the `scc` — its cost is subsumed by the `scc` annotation in the wrapper.

```
fac n = scc "fac"
        case n of Int n# -> fac# n#

fac# n# = let n = Int n#
          in
          case n of
          Int n# -> case n# of
                    0#  -> Int 1#
                    n#' -> n * fac (n-(Int 1#))
```

The subsequent optimisation of the worker includes the unfolding of the recursive call to the wrapper which contains the `scc "fac"` annotation. This `scc` is retained in the optimised version of the worker. It ensures that the entry count is incremented on each recursive call to `fac#`, as it would have been in the original definition of `fac`.

```
fac# n# = case n# of
            0#  -> Int 1#
            n#' -> case n# -# 1# of
                     n1# -> case (scc "fac" fac# n1#) of
                              Int m# -> case (n# *# m#) of
                                          r# -> Int r#
```

The presence of the scc annotation has not hindered the unboxing.

Unfortunately an scc annotation within the body of fac may still hinder the optimisation of the worker as some of the local transformations may not be possible (see Section 5.4.1). For example, annotating the subtraction in the body of fac:

```
fac n = case n of
          Int n# -> case n# of
                      0#  -> Int 1#
                      n#' -> n * fac (scc "fac-" (n-(Int 1#)))
```

results in the optimised worker:

```
fac# n# = case n# of
            0#  -> Int 1#
            n#' -> case (scc "fac-" case n# -# 1# of
                                      r1# -> Int r1#)
                     Int n1# -> case fac# n1# of
                                  Int m# -> case n# *# m# of
                                              r# -> Int r#)
```

The intervening scc annotation has prevented the case-of-case transformation being performed. In annotating the sub-expression (n-(Int 1#)) with a cost centre we are asking the profiler to measure the cost of an expression that produces a *boxed* integer. It is therefore, not entirely surprising that we may then be forced to produce the integer!

5.4.6 Foldr/Build deforestation

The foldr/build transformation was developed to remove intermediate list structures (Gill, Launchbury & Peyton Jones [1993]). The technique introduces a uniform way of constructing and consuming lists, abstracting the use of *cons* and *nil*. A simple algebraic transformation is then used to remove the intermediate lists.

Consuming Lists

A function that consumes a list in a uniform fashion can be expressed by replacing the *conses* in the list with a given function $\oplus$, and replacing the *nil* at the end of the list by a given value z. This operation is encapsulated by the higher-order function `foldr`, which can be informally defined like this:

$$\texttt{foldr}\ (\oplus)\ z\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus z)))$$

The Haskell implementation of `foldr` is:

```
foldr k z [] = z
foldr k z (x:xs) = k x (foldr f z xs)
```

Many list-consuming functions can be expressed using foldr. For example:

```
sum xs   = foldr (+) 0 xs
map f xs = foldr (\ a b -> f a : b) [] xs
```

Producing Lists

List-producing functions are similarly abstracted with respect to the *cons* and *nil* used to construct the list. For example, abstracting the above definition of `map` with respect to the : and [] used to produce the resulting list we get:

```
map f xs = build (\ c n -> foldr (\ a b -> c (f a) b) n xs)
```

The `build` function is used in the abstracted definition to supply the actual : and [] to the abstracted function.

```
build g = g (:) []
```

Foldr/Build rule

Having abstracted the *conses* and *nils* we can obtain the effect of a `foldr` consuming a list that is produced by a `build` by applying the abstracted function in the `build` directly to the *cons* and *nil* supplied to the `foldr`.

$$\boxed{\texttt{foldr}\ k\ z\ (\texttt{build}\ g) \implies g\ k\ z}$$

This is the foldr/build transformation (modulo a type restriction that ensures that the lists really are abstracted correctly (Gill, Launchbury & Peyton Jones [1993])). Consider the application of this transformation to a simple pipeline:

```
    map f (map g l)
```

$\Longrightarrow$ unfolding `map`

```
    build (\ c1 n1 -> foldr (\ a1 b1 -> c1 (f a1) b1) n1 (build
       (\ c2 n2 -> foldr (\ a2 b2 -> c2 (g a2) b2) n2 l)))
```

$\Longrightarrow$ foldr/build

```
    build (\ c1 n1 -> (\ c2 n2 -> foldr (\ a2 b2 -> c2 (g a2) b2) n2 l)
                      (\ a1 b1 -> c1 (f a1) b1) n1)
```

$\Longrightarrow$ β-reduction

```
    build (\ c1 n1 -> foldr (\ a2 b2 -> (\ a1 b1 -> c1 (f a1) b1)
                                        (g a2) b2) n1 l)
```

$\Longrightarrow$ β-reduction

```
    build (\ c1 n1 -> foldr (\ a2 b2 -> c1 (f (g a2)) b2) n1 l)
```

The result is the unfolding of `map (f.g) l` — the intermediate list has been eliminated!

Transforming `scc` annotated expressions

We now consider how the introduction of cost centres affects this transformation process. We introduce a second rule that copes with an intervening `scc` annotation around the `build`.

$$\boxed{\texttt{foldr}\ k\ z\ (\texttt{scc}\ cc\ \texttt{build}\ g) \Longrightarrow (\texttt{scc}\ cc\ g)\ k\ z}$$

This rule annotates g, the part of the result extracted from within the `scc` expression, with the cost centre. The resulting list is attributed to the enclosing cost centre since it is built by the k and z passed to g. The list built by the cost centre cc has been eliminated.

Consider the transformation of a simple pipeline with `scc` annotations:

```
    scc "mf" map f (scc "mg" map g l)
```

$\Longrightarrow$ unfolding `map`

```
    scc "mf" build
       (\ c1 n1 -> foldr (\ a1 b1 -> c1 (f a1) b1) n1 (scc "mg" build
          (\ c2 n2 -> foldr (\ a2 b2 -> c2 (g a2) b2) n2 l)))
```

$\Longrightarrow$ foldr/build with scc

```
scc "mf" build
   (\ c1 n1 -> (scc "mg"
                  (\ c2 n2 -> foldr (\ a2 b2 -> c2 (g a2) b2) n2 l))
               (\ a1 b1 -> c1 (f a1) b1) n1)
```

At this stage we would like to perform a β-reduction. Unfortunately this will unfold the function `(\ a1 b1 -> c1 (f a1) b1)` within the inner `scc`. We can only proceed if the enclosing cost centre is known.

$\Longrightarrow$ β-reduction with annotation

```
scc "mf" build
   (\ c1 n1 -> (scc "mg"
                foldr (\ a2 b2 -> (scc_sub "mf"
                                     (\ a1 b1 -> c1 (f a1) b1))
                                  (g a2) b2) n1 l))
```

$\Longrightarrow$ β-reduction with annotation

```
scc "mf" build
   (\ c1 n1 -> (scc "mg"
                foldr (\ a2 b2 -> (scc_sub "mf"
                                    c1 (f (scc_sub "mg" g a2)) b2))
                      n1 l))
```

The resulting expression is the same but the cost centre annotations are somewhat mystifying. Let us consider what costs are now attributed where:

- `"mg"` is attributed with the mapped function `g`. It is also attributed with the costs of consuming `l` — the application of `foldr` to `l`.
- `"mf"` is attributed with the function `f` and the construction of the new list — the *cons* and *nil* embedded within the outer `build`.
- If the example is extended with a third pipeline element, the middle element incurs no list consumption or list production costs — both intermediate lists are eliminated.

This attribution of costs seems quite appropriate.

If the enclosing cost centre, `"mf"`, was not known at compile time the β-reductions above would not be possible since the unfolded expression could not be annotated with the enclosing cost centre. Though this would reduce the opportunities for further optimisation, the intermediate list is still eliminated since the foldr/build transformation has been applied.

5.4.7 Transformation of evaluation scoping

So far we have been looking at program transformation that preserves lexical cost attribution. We now turn our attention to the differences encountered when preserving the evaluation and hybrid cost attributions.

The preservation of evaluation cost attribution places some additional restrictions on program transformation. Since evaluation scoping attributes the costs of a applying a λ-abstraction to the scope of the application site, we have to ensure that the cost centre enclosing the application site is preserved. For example, the foldr/build `scc` transformation

$$\texttt{foldr}\ k\ z\ (\texttt{scc}\ cc\ \texttt{build}\ g) \implies (\texttt{scc}\ cc\ g)\ k\ z$$

is not possible because the application site of k is been moved from the body of `foldr`, in the scope of the enclosing cost centre, to the body of g in the scope of cc.

As our preference is for lexical profiling we do not explore this issue further.

5.4.8 Transformation of hybrid profiling

Preservation of hybrid cost attribution requires us to treat $\texttt{scc}_{caf}$ and $\texttt{scc}_{dict}$ boundaries as evaluation scoping `scc` boundaries. All other `scc` boundaries are treated as for lexical scoping.

In practice this causes us no problems, since we only introduce the $\texttt{scc}_{caf}$ and $\texttt{scc}_{dict}$ annotations after all the compiler optimisations have been performed. The restrictions during optimisation are identical to those described for lexical profiling.

5.5 Profiled Execution

Having optimised the program, being sure to maintain the appropriate cost attribution, we then have to generate code to execute the program. When profiling we must arrange for the profiled execution events, such as a timer interrupt or heap allocation, to be attributed to the appropriate cost centre. This task appears particularly difficult in a lazy implementation since the execution of a particular expression may be interleaved with the execution of other expressions (Section 3.2.4).

Fortunately the abstract cost semantics of Section 5.5.2 suggest an implementation.

This implementation has two major components:

- The cost centre in the context of the expression currently being evaluated is stored in a special *current cost centre* register.[1] This register is also used to return the result cost centre, cc_z, with the result of an expression. As costs are incurred they are attributed to the *current cost centre.*

- Every heap-allocated closure has an extra field that identifies the cost centre responsible for allocating it — the *allocating cost centre.* The current cost centre is stored in each closure when it is allocated.

These two ideas are common to the implementation of all the profiling schemes. The differences between the implementations are in the way the cost centres are manipulated during execution. Namely:

- When the current cost centre is loaded, and

- When the current cost centre is saved and restored.

The actual value of the current cost centre at a particular point during the execution will differ if the different semantics require the costs of the expression currently being evaluated to be attributed to different cost centres.

Unfortunately there is a large gap between the abstract cost semantics of Section 4.2 and any implementation based on graph reduction. This is because the abstract cost semantics uses the *eval-apply* model of reduction while graph reduction uses a *push-enter* model of reduction (Peyton Jones [1992, Section 3.2]). The difference between the two models of reduction is in the treatment of function application. The eval-apply model evaluates the λ-abstraction being applied and then evaluates the body. In contrast, the push-enter model pushes the argument being applied on an argument stack and tail-calls (or *enters*) the function expression. When the evaluation of the function is complete the argument is retrieved from the stack and the body evaluated without returning the λ-abstraction.

[1] Our *current cost centre* is very similar to the *current function* used in the New Jersey SML profiler (Appel, Duba & MacQueen [1988]). Their work was motivated by the need to keep track of interleaved execution arising from transforming optimisations. Though this is also an issue for us: we arrived at our scheme because we can't avoid the interleaving arising from lazy evaluation.

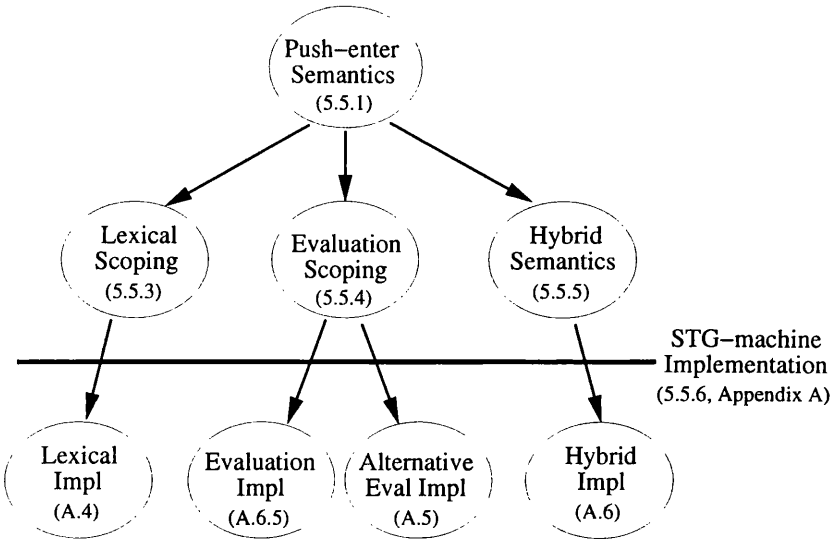


Figure 5.6: Development of push-enter semantics and STG implementation

Rather than trying to describe our implementation directly in terms of the eval-apply semantics we introduce an abstract push-enter semantics (Section 5.5.1). We then augment this semantics with notions of cost and cost attribution (Section 5.5.2) and present complete cost augmented push-enter semantics for each of the profiling schemes (Sections 5.5.3, 5.5.4 and 5.5.5). This allows us to highlight the differences between the *eval-apply* and *push-enter* models without being caught up with the details of our implementation. Indeed, up to this point the discussion applies to any graph-reduction based implementation e.g. the G-machine (Augustsson & Johnsson [1989]) the TIM (Fairbairn & Wray [1987]), and the STG-machine (Peyton Jones [1992]).

Finally, we discuss the details of our STG-machine implementation (Section 5.5.6). The STG-level manipulation of cost centres is made precise by extending the STG-machine state-transition semantics with the manipulation of cost centres (Appendix A). The development of the different push-enter semantics and our STG-machine implementations is summarised in Figure 5.6.

5.5.1 Push-enter reduction semantics

The push-enter semantics are given for the same restricted language as the eval-apply semantics of Section 4.2.1. The reduction rules are presented in Figure 5.7. We have

presented these semantics in the same style as the eval-apply semantics. The only new component of the semantics is the argument stack — an ordered sequence. () denotes the empty stack and $a : as$ denotes the stack obtained by pushing the argument a onto the front of top stack as.

$$\begin{array}{lcll} as & \in & Stack & ::= & a : as \\ & & & \mid & () \end{array}$$

A judgement has the form $\Gamma, as : e \Downarrow \Delta : z$ which should be read: "the term e in the context of the set of bindings Γ and argument stack as reduces to the value z together with the (modified) set of bindings Δ." During the course of evaluation the argument stack is consumed by the expression being evaluated and the heap may be extended with new bindings and/or have old bindings updated with their results.

It is important to note that a λ-abstraction is returned if and only if there are no arguments available to apply. This is where the two evaluation models differ. In the eval-apply semantics there is no argument stack. A λ-abstraction is always returned to the application site before being applied.

Reduction rules

Referring to the rules in Figure 5.7, the main difference between the eval-apply semantics and the push-enter semantics is in the rules for application and λ-abstractions. The *App* rule *enters* the function expression e in a context with the argument x pushed onto the argument stack. This argument is applied by the Lam_{Arg} rule once e has been reduced to a λ-abstraction — the λ-abstraction is not returned.

The *Lambda* rule has two cases depending if there is an argument available on the argument stack. If the argument stack is non-empty the Lam_{Arg} rule substitutes the argument on the top of the stack for the λ-variable in the body of the λ-abstraction and enters the body in a context containing the remaining arguments. If the argument stack is empty the Lam_{NoArg} rule simply returns the λ-abstraction value.

For convenience we split the *Variable* rule into two cases. If the bound expression is already in whnf, i.e. $x \mapsto z$, the bound expression z does not need to be evaluated and no update is required. This is captured by the Var_{WHNF} rule which just enters a renamed copy of the bound value $\widehat{z}$ with the same context. Alternatively the bound expression

$$\frac{\Gamma, (x : as) : e \ \Downarrow \ \Delta : z}{\Gamma, as : e \ x \ \Downarrow \ \Delta : z} \qquad App$$

$$\frac{\Gamma, as : e[a/x] \ \Downarrow \ \Delta : z}{\Gamma, (a : as) : \lambda x.e \ \Downarrow \ \Delta : z} \qquad Lam_{Arg}$$

$$\Gamma, () : \lambda x.e \ \Downarrow \ \Gamma : \lambda x.e \qquad Lam_{NoArg}$$

$$\frac{\{\Gamma, x \mapsto z\}, as : \widehat{z} \ \Downarrow \ \Delta : z'}{\{\Gamma, x \mapsto z\}, as : x \ \Downarrow \ \Delta : z'} \qquad Var_{WHNF}$$

$$\frac{\Gamma, () : e \ \Downarrow \ \Delta : z \qquad \{\Delta, x \mapsto z\}, as : \widehat{z} \ \Downarrow \ \Theta : z'}{\{\Gamma, x \mapsto e\}, as : x \ \Downarrow \ \Theta : z'} \qquad Var_{Thunk}$$

$$\frac{\{\Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}, as : e \ \Downarrow \ \Delta : z}{\Gamma, as : \texttt{let } x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n \texttt{ in } e \ \Downarrow \ \Delta : z} \qquad Let$$

$$\Gamma, () : C \ x_1 \cdots x_n \ \Downarrow \ \Gamma : C \ x_1 \cdots x_n \qquad Constr$$

$$\frac{\Gamma, () : e \ \Downarrow \ \Delta : C_k \ x_1 \cdots x_{m_k} \qquad \Delta, as : e_k[x_i/y_i]_{i=1}^{m_k} \ \Downarrow \ \Theta : z}{\Gamma, as : \texttt{case } e \texttt{ of } \{C_i \ y_1 \cdots y_{m_i} \texttt{ -> } e_i\}_{i=1}^{n} \ \Downarrow \ \Theta : z} \qquad Case$$

$$\frac{\Gamma, () : e_1 \ \Downarrow \ \Delta : z_1 \qquad \Delta, () : e_2 \ \Downarrow \ \Theta : z_2}{\Gamma, () : e_1 \oplus e_2 \ \Downarrow \ \Theta : z_1 \oplus z_2} \qquad Prim$$

Figure 5.7: Push-Enter Reduction Rules

may not be in whnf. In this case the bound expression e must be evaluated and the binding updated with its result. The Var_{Thunk} rule evaluates the bound expression in a context with x omitted from the heap (to detect cyclic data dependencies) and an empty argument stack. The empty argument stack ensures that the result of the expression is returned before being applied to any arguments. If a result value z is returned the heap is updated with the binding $x \mapsto z$ and a renamed version of the result $\widehat{z}$ is entered with the arguments as now available in the context.

The remaining rules are basically the same as the eval-apply rules, except for the addition of the argument stack. The *Let* rule extends the heap with the new bindings and evaluates the body e, in the extended context. Renaming ensures that there are no name clashes. The *Constr* rule always returns the constructor value. A constructor always has an empty argument stack as it cannot be applied. The *Case* rule reduces the body e in a context with no arguments. When the constructor is returned the appropriate alternative is selected; the constructor arguments are substituted; and the alternative is entered with the original argument stack in the context. The *Case* rule only succeeds if the constructor returned is contained in the alternatives.

Finally the rule for primitive applications. As a primitive application always returns a constructor the argument stack is always empty. The *Prim* rule evaluates each constructor argument in a context with an empty stack, computes the result and returns it.

Reduction sequences

The reduction sequences are extended with an argument stack. If $\Gamma, as : e \Downarrow \Delta : z$ we write:

$$
\begin{array}{l}
\{\Gamma\}, as : e \\
\quad\Big|\ \text{a sub-proof} \\
\quad\Big|\ \text{another sub-proof} \\
\{\Delta\} : z
\end{array}
$$

For example, the push-enter reduction sequence for the expression `let` f=$\lambda x.x$+1 `in` $f\ 3$ would be written:

$\{\Delta\}, () : \texttt{let } f{=}\lambda x.x{+}1 \texttt{ in } f\ 3$ *Let*
$\quad \{\Delta, f \mapsto \lambda x.x{+}1\}, () : f\ 3$ *App*
$\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\}, (3 : ()) : f$ Var_{WHNF}
$\quad\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\}, (3 : ()) : \lambda x_1.x_1{+}1$ Lam_{Arg}
$\quad\quad\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\}, () : 3{+}1$ *Prim*
$\quad\quad\quad\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\}, () : 3$ *Constr*
$\quad\quad\quad\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\} : 3$
$\quad\quad\quad\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\}, () : 1$ *Constr*
$\quad\quad\quad\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\} : 1$
$\quad\quad\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\} : 4$ (evaluate +)
$\quad\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\} : 4$
$\quad\quad \{\Delta, f \mapsto \lambda x.x{+}1\} : 4$
$\quad \{\Delta, f \mapsto \lambda x.x{+}1\} : 4$
$\{\Delta, f \mapsto \lambda x.x{+}1\} : 4$

Comparison with the equivalent eval-enter reduction sequence on page 43 reveals that the nesting is much deeper with the push-enter semantics since the λ-abstraction is not returned to the application. Fortunately this deeper nesting is not a problem for the implementation as all reduction steps that require a single sub-proof can be implemented with a tail-call.

5.5.2 Cost-augmented push-enter semantics

We now extend the push-enter reduction semantics with a notion of cost and cost attribution. These extensions are identical to those required for the eval-apply cost semantics in Section 4.2.2. We introduce: a new `scc` language construct; a cost attribution θ, mapping cost centres to integers, with combining operator $\uplus$; cost centre annotations on the variable/expression pairs in the heap; the same initial heap bindings, Γ_{init}; and extend the form of the judgement with enclosing and returning cost centres.

The augmented judgement form is $cc, \Gamma, as : e \Downarrow_\theta \Delta : z, cc_z$ which should be read: "the term e in the context of the set of (annotated) bindings Γ, argument stack as and enclosing cost centre cc, reduces to the value z together with the (modified) set of (annotated) bindings Δ and result cost centre cc_z, attributing costs θ." The result cost centre, cc_z, is the cost centre that enclosed the expression that declared or constructed the result value z.

Push-enter reduction sequences are similarly extended with the enclosing and returned

cost centre, and the reported cost attribution if required.

Finally, we introduce the same set of reduction costs R_λ, R_C, H, V, U, A, C, and P. This allows us to compare the abstract cost attribution, θ, reported by the push-enter semantics with that reported by the corresponding eval-apply semantics. Apart from a reduction in the number of λ-abstraction returns (R_λ) the cost attributions should be equivalent. However, the actual costs involved in a push-enter implementation would be different from the actual costs of an eval-apply implementation.

We present and discuss the push-enter semantics for each of our profiling schemes in the following sections.

5.5.3 Lexical scoping

Mapping the lexical scoping eval-apply semantics onto the push-enter semantics is quite straightforward. In the eval-apply semantics the application rule for lexical scoping, App_{lex} (Section 4.2.4), attributes the evaluation of the body of the λ-abstraction to the cost centre that is returned with the λ-abstraction. In the push-enter semantics the λ-abstraction evaluates its body directly without returning. The cost of evaluating the body of the λ-abstraction are attributed to the cost centre of the λ-abstraction.

A summary of the cost centre manipulation for the push-enter semantics is contrasted with the eval-apply manipulation in Figure 5.8. The cost augmented push-enter reduction rules for lexical scoping are given Figure 5.9.

The *App* rule enters the function expression e with the argument pushed on the stack and the enclosing cost centre cc in the context. We do not associate any cost with the *App* rule. The cost of a curried application, A, is associated with the Lam_{Arg} rule when the λ-abstraction is actually applied.

The Lam_{Arg} rule evaluates the body of the λ-abstraction in the context of the cost centre enclosing the λ-abstraction. The cost of the application, A, is also attributed to the cost centre cc. This corresponds directly to the attribution of costs achieved by the eval-apply App_{lex} rule, except that the cost of returning the λ-abstraction, R_λ is no longer incurred.

The Lam_{NoArg} rule is identical to the eval-apply *Lambda* rule. It returns the λ-abstraction with the enclosing cost centre cc attached. The cost of returning the λ-abstraction R_λ is attributed to the cost centre cc.

Execution Event	Eval-Apply Semantics	Push-Enter Semantics
Application	Evaluate λ-abs with *cc* of application and applied body with *cc* of λ-abs (returned)	Push argument and *enter* λ-abs expression with *cc* of application
Apply λ-abs	n.a. (see Application)	Evaluate body with *cc* of λ-abs
Return λ-abs	Return *cc* of λ-abs	Return *cc* of λ-abs
Evaluate thunk	Set *cc* of thunk	Set *cc* of thunk
Apply thunk result	n.a. (see Application)	Set *cc* of λ-abs (returned)
Return thunk result	Return result *cc*	Return result *cc*
Return constr	Return *cc* of constr	Return *cc* of constr
Case	Restore enclosing *cc* for evaluation of alternative	Restore enclosing *cc* for evaluation of alternative
Evaluate SCC	Set *cc* of `scc`	Set *cc* of `scc`

Figure 5.8: Summary of Lexical Scoping Cost Centre Manipulation

The Var_{WHNF} rule enters the (renamed) bound value $\hat{z}$, attributing the costs of evaluation to the bound cost centre cc_z. This ensures that the Lam_{Arg} rule attributes the costs of applying a λ-abstraction value and evaluating its body to the cost centre that declared the λ-abstraction. The only exception to this is top-level functions which have a `"SUB"` cost centre attached. Their evaluation costs are attributed to the demanding cost centre *cc*. This choice is captured by the SUB_l selector. The cost of entering the variable is attributed to the demanding cost centre *cc*.

The Var_{Thunk} rule attributes the evaluation of the bound expression e to its whnf to the bound cost centre cc_e. The renamed result $\hat{z}$ is then entered in the context of the returned cost centre cc_z. Again, this ensures that the costs of applying a λ-abstraction value and evaluating its body are attributed to the declaring cost centre returned by the Lam_{NoArg} rule. Before entering the result the heap binding is updated with the result z annotated with the result cost centre cc_z. This ensures that subsequent enters of x also attribute evaluation to the declaring cost centre cc_z. The cost of entering the variable, V, is attributed to the demanding cost centre *cc* and the cost of the update, U, is attributed to the result cost centre cc_z.

The remaining reduction rules hold no surprises. The cost centre manipulation is

$$\frac{cc, \Gamma, (x : as) : e \;\Downarrow_\theta\; \Delta : z, cc_z}{cc, \Gamma, as : e\ x \;\Downarrow_\theta\; \Delta : z, cc_z} \quad App$$

$$\frac{cc, \Gamma, as : e[a/x] \;\Downarrow_\theta\; \Delta : z, cc_z}{cc, \Gamma, (a : as) : \lambda x.e \;\Downarrow_{\{cc \mapsto \mathrm{A}\} \uplus \theta}\; \Delta : z, cc_z} \quad Lam_{Arg}$$

$$cc, \Gamma, () : \lambda x.e \;\Downarrow_{\{cc \mapsto \mathrm{R}_\lambda\}}\; \Gamma : \lambda x.e, cc \quad Lam_{NoArg}$$

$$\frac{\mathrm{SUB}_l(z, cc_z, cc), \{\Gamma, x \overset{cc_x}{\mapsto} z\}, as : \hat{z} \;\Downarrow_\theta\; \Delta : z', cc_{z'}}{cc, \{\Gamma, x \overset{cc_x}{\mapsto} z\}, as : x \;\Downarrow_{\{cc \mapsto \mathrm{V}\} \uplus \theta}\; \Delta : z', cc_{z'}} \quad Var_{WHNF}$$

$$\frac{cc_e, \Gamma, () : e \;\Downarrow_{\theta_1}\; \Delta : z, cc_z \qquad cc_z, \{\Delta, x \overset{cc_x}{\mapsto} z\}, as : \hat{z} \;\Downarrow_{\theta_2}\; \Theta : z', cc_{z'}}{cc, \{\Gamma, x \overset{cc_x}{\mapsto} e\}, as : x \;\Downarrow_{\{cc \mapsto \mathrm{V}\} \uplus \{cc_x \mapsto \mathrm{U}\} \uplus \theta_1 \uplus \theta_2}\; \Theta : z', cc_{z'}} \quad Var_{Thunk}$$

$$\frac{cc, \{\Gamma, x_1 \overset{cc}{\mapsto} e_1, \ldots, x_n \overset{cc}{\mapsto} e_n\}, as : e \;\Downarrow_\theta\; \Delta : z, cc_z}{cc, \Gamma, as : \texttt{let}\ x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n\ \texttt{in}\ e \;\Downarrow_{\{cc \mapsto n*\mathrm{H}\} \uplus \theta}\; \Delta : z, cc_z} \quad Let$$

$$cc, \Gamma, () : C\ x_1 \cdots x_n \;\Downarrow_{\{cc \mapsto \mathrm{R}_C\}}\; \Gamma : C\ x_1 \cdots x_n, cc \quad Constr$$

$$\frac{cc, \Gamma, () : e \;\Downarrow_{\theta_1}\; \Delta : C_k\ x_1 \cdots x_{m_k}, cc_C \qquad cc, \Delta, as : e_k[x_i/y_i]_{i=1}^{m_k} \;\Downarrow_{\theta_2}\; \Theta : z, cc_z}{cc, \Gamma, as : \texttt{case}\ e\ \texttt{of}\ \{C_i\ y_1 \cdots y_{m_i}\ \texttt{->}\ e_i\}_{i=1}^{n} \;\Downarrow_{\{cc \mapsto \mathrm{C}\} \uplus \theta_1 \uplus \theta_2}\; \Theta : z, cc_z} \quad Case$$

$$\frac{cc, \Gamma, () : e_1 \;\Downarrow_{\theta_1}\; \Delta : z_1, cc_1 \qquad cc, \Delta, () : e_2 \;\Downarrow_{\theta_2}\; \Theta : z_2, cc_2}{cc, \Gamma, () : e_1 \oplus e_2 \;\Downarrow_{\{cc \mapsto \mathrm{P}\} \uplus \theta_1 \uplus \theta_2}\; \Theta : z_1 \oplus z_2, cc} \quad Prim$$

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

where $\mathrm{SUB}_l(\lambda x.e, \texttt{"SUB"}, cc) = cc$
$\mathrm{SUB}_l(\quad z, \quad cc_z, cc) = cc_z$

Figure 5.9: Lexical Scoping Push-Enter Reduction Rules

identical to the corresponding eval-apply reduction rule in Figure 4.5.

5.5.4 Evaluation scoping

Mapping the evaluation scoping eval-apply semantics onto the push-enter semantics is not so straight forward. In the eval-apply semantics the application rule for evaluation scoping, App_{eval} (Section 4.2.3), attributes the evaluation of the body of the λ-abstraction to the cost centre enclosing the application site. In the corresponding push-enter semantics the cost centre which enclosed the application site must be available when the Lam_{Arg} rule is applied since the cost of evaluating the body of the λ-abstraction must be attributed to this cost centre.

One way to achieve this is to push both the argument and the cost centre of the application site, (cc, a), onto the argument stack in the *App* rule. The Lam_{Arg} rule can then set this cost centre when the λ-abstraction is applied.

$$\frac{cc, \Gamma, ((cc, x) : as) : e \ \Downarrow_\theta \ \Delta : z, cc_z}{cc, \Gamma, as : e\ x \ \Downarrow_\theta \ \Delta : z, cc_z} \quad App$$

$$\frac{cc_{app}, \Gamma, as : e[a/x] \ \Downarrow_\theta \ \Delta : z, cc_z}{cc, \Gamma, ((cc_{app}, a) : as) : \lambda x.e \ \Downarrow_{\{cc \mapsto A\} \uplus \theta} \ \Delta : z, cc_z} \quad Lam_{Arg}$$

All the other push-enter reduction rules are identical to the rules for lexical scoping in Figure 5.9.

Though pushing a cost centre onto the stack with each argument seems quite straight forward it is very intrusive in our STG-machine implementation since the stacks have a non-trivial structure. It also seems excessive when one considers that the STG-machine introduces multiple argument λ-abstractions that grab all their arguments off the stack in one go. The only cost centre that is actually required is the cost centre associated with the last argument retrieved from the stack.

In the light of this we have developed an alternative evaluation semantics that does not require the cost centre of the application site to be pushed on the stack with the argument. Instead, it ensures that the cost centre of the application site encloses the λ-abstraction when the Lam_{Arg} reduction rule is applied. This is achieved by insisting that all sub-reduction sequences that evaluate an expression in the context of a different cost centre ensure that the result is returned even if an argument is available on the stack.

Execution Event	Eval-Apply Semantics	Push-Enter Semantics
Application	Evaluate λ-abs and applied body with *cc* of application	Push argument and *enter* λ-abs expression with *cc* of application
Apply λ-abs	n.a. (see Application)	Evaluate body with enclosing *cc* — the *cc* of the application
Return λ-abs	Return *cc* of λ-abs	Return *cc* of λ-abs
Evaluate thunk	Set *cc* of thunk	Set *cc* of thunk
Apply thunk result	n.a. (see Application)	Restore enclosing *cc*
Return thunk result	Return result *cc*	Return result *cc*
Return constr	Return *cc* of constr	Return *cc* of constr
Case	Restore enclosing *cc* for evaluation of alternative	Restore enclosing *cc* for evaluation of alternative
Evaluate SCC	Set *cc* of `scc`	Set *cc* of `scc`
Apply SCC result	n.a. (see Application)	Restore enclosing *cc*
Return SCC result	Return result *cc*	Return result *cc*

Figure 5.10: Summary of Evaluation Scoping Cost Centre Manipulation

This is achieved by evaluating the expression in the context of an empty argument stack. If the returned result is a λ-abstraction and an argument is available the cost centre of the application site is restored and the λ-abstraction entered with the argument now available on the stack. If the result is not a λ-abstraction or no argument is available the the result is returned without restoring the cost centre.

A summary of the cost centre manipulation for this evaluation semantics is contrasted with the eval-apply manipulation in Figure 5.10.

The cost augmented push-enter reduction rules are given in Figure 5.11. Most of the rules are identical to the lexical scoping rules (Figure 5.9) described in Section 5.5.3. The rules that differ are the Var_{WHNF}, Var_{Thunk} and SCC rules which have to ensure that the cost centre of the application site is loaded/restored before a λ-abstraction value is entered with the argument available on the stack.

The Var_{WHNF} enters the renamed value $\widehat{z}$. The $\text{SUB}_e(z, as, cc_z, cc)$ selector determines the cost centre. There are two cases:

- A λ-abstraction that is applied by the Lam_{Arg} rule. Under evaluation scoping the costs of evaluating the body of the λ-abstraction should be attributed to the cost

$$\frac{cc, \Gamma, (x : as) : e \ \Downarrow_\theta \ \Delta : z, cc_z}{cc, \Gamma, as : e\ x \ \Downarrow_\theta \ \Delta : z, cc_z} \quad App$$

$$\frac{cc, \Gamma, as : e[a/x] \ \Downarrow_\theta \ \Delta : z, cc_z}{cc, \Gamma, (a : as) : \lambda x.e \ \Downarrow_{\{cc \mapsto \mathrm{A}\} \uplus \theta} \ \Delta : z, cc_z} \quad Lam_{Arg}$$

$$cc, \Gamma, () : \lambda x.e \ \Downarrow_{\{cc \mapsto \mathrm{R}_\lambda\}} \ \Gamma : \lambda x.e, cc \quad Lam_{NoArg}$$

$$\frac{\mathrm{SUB}_e(z, as, cc_z, cc), \{\Gamma, x \overset{cc_z}{\mapsto} z\}, as : \widehat{z} \ \Downarrow_\theta \ \Delta : z', cc_{z'}}{cc, \{\Gamma, x \overset{cc_z}{\mapsto} z\}, as : x \ \Downarrow_{\{cc \mapsto \mathrm{V}\} \uplus \theta} \ \Delta : z', cc_{z'}} \quad Var_{WHNF}$$

$$\frac{cc_e, \Gamma, () : e \ \Downarrow_{\theta_1} \ \Delta : z, cc_z \quad \mathrm{SUB}_e(z, as, cc_z, cc), \{\Delta, x \overset{cc_z}{\mapsto} z\}, as : \widehat{z} \ \Downarrow_{\theta_2} \ \Theta : z', cc_{z'}}{cc, \{\Gamma, x \overset{cc_z}{\mapsto} e\}, as : x \ \Downarrow_{\{cc \mapsto \mathrm{V}\} \uplus \{cc_z \mapsto \mathrm{U}\} \uplus \theta_1 \uplus \theta_2} \ \Theta : z', cc_{z'}} \quad Var_{Thunk}$$

$$\frac{cc, \{\Gamma, x_1 \overset{cc}{\mapsto} e_1, \ldots, x_n \overset{cc}{\mapsto} e_n\}, as : e \ \Downarrow_\theta \ \Delta : z, cc_z}{cc, \Gamma, as : \texttt{let } x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n \texttt{ in } e \ \Downarrow_{\{cc \mapsto n*\mathrm{H}\} \uplus \theta} \ \Delta : z, cc_z} \quad Let$$

$$cc, \Gamma, () : C\ x_1 \cdots x_n \ \Downarrow_{\{cc \mapsto \mathrm{R}_C\}} \ \Gamma : C\ x_1 \cdots x_n, cc \quad Constr$$

$$\frac{cc, \Gamma, () : e \ \Downarrow_{\theta_1} \ \Delta : C_k\ x_1 \cdots x_{m_k}, cc_C \quad cc, \Delta, as : e_k[x_i/y_i]_{i=1}^{m_k} \ \Downarrow_{\theta_2} \ \Theta : z, cc_z}{cc, \Gamma, as : \texttt{case } e \texttt{ of } \{C_i\ y_1 \cdots y_{m_i} \texttt{ -> } e_i\}_{i=1}^{n} \ \Downarrow_{\{cc \mapsto \mathrm{C}\} \uplus \theta_1 \uplus \theta_2} \ \Theta : z, cc_z} \quad Case$$

$$\frac{cc, \Gamma, () : e_1 \ \Downarrow_{\theta_1} \ \Delta : z_1, cc_1 \quad cc, \Delta, () : e_2 \ \Downarrow_{\theta_2} \ \Theta : z_2, cc_2}{cc, \Gamma, () : e_1 \oplus e_2 \ \Downarrow_{\{cc \mapsto \mathrm{P}\} \uplus \theta_1 \uplus \theta_2} \ \Theta : z_1 \oplus z_2, cc} \quad Prim$$

$$\frac{cc_{scc}, \Gamma, () : e \ \Downarrow_{\theta_1} \ \Delta : z, cc_z \quad \mathrm{SUB}_e(z, as, cc_z, cc), \Delta, as : z \ \Downarrow_{\theta_2} \ \Theta : z', cc_{z'}}{cc, \Gamma, as : \texttt{scc } cc_{scc} \ e \ \Downarrow_{\theta_1 \uplus \theta_2} \ \Theta : z', cc_{z'}} \quad SCC$$

where
$$\mathrm{SUB}_e(\lambda x.e, \ as, \texttt{"SUB"}, cc) = cc$$
$$\mathrm{SUB}_e(\lambda x.e, a : as, \ cc_z, cc) = cc$$
$$\mathrm{SUB}_e(\ z, \ as, \ cc_z, cc) = cc_z$$

Figure 5.11: Evaluation Scoping Push-Enter Reduction Rules

centre enclosing the application site. We ensure that any intervening reduction rules that modify this enclosing cost centre do not supply the argument, forcing the result to be returned and allowing the enclosing cost centre (the cost centre of the application site) to be restored before the result is entered with the argument available on the stack. Thus, the costs of applying the λ-abstraction and evaluating its body can be attributed to the enclosing cost centre cc.

- A value that is by the Lam_{NoArg} or *Constr* rule. The value is returned with its cost centre cc_z, unless it is a subsumed λ-abstraction. This returned cost centre is attached to any updated closures.

The $\text{SUB}_e(z, as, cc_z, cc)$ selector selects the enclosing cost centre cc if the value z is a λ-abstraction and the bound cost centre is "SUB" or the argument stack is non-empty, otherwise the bound cost centre cc_z is selected.

The Var_{Thunk} rule evaluates the bound expression e in the context of the cost centre cc_e without supplying any arguments. If the result is a λ-abstraction and the argument stack is non-empty the cost centre of the application site is restored before the λ-abstraction is entered with the argument supplied. If no argument is available the value is entered in the context of the result cost centre cc_z, and returned again. This is the same choice as the Var_{WHNF} rule (except that the "SUB" case is redundant). The same $\text{SUB}_e(z, as, cc_z, cc)$ selector is used.

The *SCC* rule evaluates the annotated expression e in the context of the cost centre cc_{scc}. Under evaluation scoping it must restore the cost centre of the application site before supplying any arguments. It evaluates the annotated expression e without supplying any arguments. If the result is a λ-abstraction and the argument stack is non-empty the cost centre of the application site is restored before the λ-abstraction is entered with the argument supplied. Again this is captured by the $\text{SUB}_e(z, as, cc_z, cc)$ selector.

5.5.5 Hybrid profiling scheme

The semantic rules for our hybrid profiling scheme are given in Figure 5.12. These are identical to the rules for evaluation scoping (Figure 5.11) described in Section 5.5.4, except that the $\text{SUB}_h(z, as, cc_z, cc)$ selector only selects/restores the cost centre of the application site if the λ-abstraction is subsumed or declared in a CAF or dictionary i.e. the

$$\frac{cc, \Gamma, (x : as) : e \ \Downarrow_{\theta} \ \Delta : z, cc_z}{cc, \Gamma, as : e \ x \ \Downarrow_{\theta} \ \Delta : z, cc_z} \quad App$$

$$\frac{cc, \Gamma, as : e[a/x] \ \Downarrow_{\theta} \ \Delta : z, cc_z}{cc, \Gamma, (a : as) : \lambda x.e \ \Downarrow_{\{cc \mapsto \mathrm{A}\} \uplus \theta} \ \Delta : z, cc_z} \quad Lam_{Arg}$$

$$cc, \Gamma, () : \lambda x.e \ \Downarrow_{\{cc \mapsto \mathrm{R}_{\lambda}\}} \ \Gamma : \lambda x.e, cc \quad Lam_{NoArg}$$

$$\frac{\mathrm{SUB}_h(z, as, cc_z, cc), \{\Gamma, x \overset{cc_z}{\mapsto} z\}, as : \widehat{z} \ \Downarrow_{\theta} \ \Delta : z', cc_{z'}}{cc, \{\Gamma, x \overset{cc_z}{\mapsto} z\}, as : x \ \Downarrow_{\{cc \mapsto \mathrm{V}\} \uplus \theta} \ \Delta : z', cc_{z'}} \quad Var_{WHNF}$$

$$\frac{cc_e, \Gamma, () : e \ \Downarrow_{\theta_1} \ \Delta : z, cc_z \quad \mathrm{SUB}_h(z, as, cc_z, cc), \{\Delta, x \overset{cc_z}{\mapsto} z\}, as : \widehat{z} \ \Downarrow_{\theta_2} \ \Theta : z', cc_{z'}}{cc, \{\Gamma, x \overset{cc_e}{\mapsto} e\}, as : x \ \Downarrow_{\{cc \mapsto \mathrm{V}\} \uplus \{cc_z \mapsto \mathrm{U}\} \uplus \theta_1 \uplus \theta_2} \ \Theta : z', cc_{z'}} \quad Var_{Thunk}$$

$$\frac{cc, \{\Gamma, x_1 \overset{cc}{\mapsto} e_1, \ldots, x_n \overset{cc}{\mapsto} e_n\}, as : e \ \Downarrow_{\theta} \ \Delta : z, cc_z}{cc, \Gamma, as : \texttt{let } x_1\texttt{=}e_1, \ldots, x_n\texttt{=}e_n \texttt{ in } e \ \Downarrow_{\{cc \mapsto n*\mathrm{H}\} \uplus \theta} \ \Delta : z, cc_z} \quad Let$$

$$cc, \Gamma, () : C \ x_1 \cdots x_n \ \Downarrow_{\{cc \mapsto \mathrm{R}_C\}} \ \Gamma : C \ x_1 \cdots x_n, cc \quad Constr$$

$$\frac{cc, \Gamma, () : e \ \Downarrow_{\theta_1} \ \Delta : C_k \ x_1 \cdots x_{m_k}, cc_C \quad cc, \Delta, as : e_k[x_i/y_i]_{i=1}^{m_k} \ \Downarrow_{\theta_2} \ \Theta : z, cc_z}{cc, \Gamma, as : \texttt{case } e \texttt{ of } \{C_i \ y_1 \cdots y_{m_i} \texttt{ -> } e_i\}_{i=1}^{n} \ \Downarrow_{\{cc \mapsto \mathrm{C}\} \uplus \theta_1 \uplus \theta_2} \ \Theta : z, cc_z} \quad Case$$

$$\frac{cc, \Gamma, () : e_1 \ \Downarrow_{\theta_1} \ \Delta : z_1, cc_1 \quad cc, \Delta, () : e_2 \ \Downarrow_{\theta_2} \ \Theta : z_2, cc_2}{cc, \Gamma, () : e_1 \oplus e_2 \ \Downarrow_{\{cc \mapsto \mathrm{P}\} \uplus \theta_1 \uplus \theta_2} \ \Theta : z_1 \oplus z_2, cc} \quad Prim$$

$$\frac{cc_{scc}, \Gamma, () : e \ \Downarrow_{\theta_1} \ \Delta : z, cc_z \quad \mathrm{SUB}_h(z, as, cc_z, cc), \Delta, as : z \ \Downarrow_{\theta_2} \ \Theta : z', cc_{z'}}{cc, \Gamma, as : \texttt{scc } cc_{scc} \ e \ \Downarrow_{\theta_1 \uplus \theta_2} \ \Theta : z', cc_{z'}} \quad SCC$$

where

$$\begin{aligned}
\mathrm{SUB}_h(\lambda x.e, \quad as, \ \texttt{"SUB"}, cc) &= cc \\
\mathrm{SUB}_h(\lambda x.e, a : as, \ \texttt{"CAF"}, cc) &= cc \\
\mathrm{SUB}_h(\lambda x.e, a : as, \texttt{"DICT"}, cc) &= cc \\
\mathrm{SUB}_h(\quad z, \quad as, \quad cc_z, cc) &= cc_z
\end{aligned}$$

Figure 5.12: Hybrid Push-Enter Reduction Rules

λ-abstraction's cost centre cc_z is "SUB", "CAF" or "DICT". The costs of applying all other λ-abstractions are attributed to the cost centre of the declaration site, not the application site.

5.5.6 STG-machine implementations

We now outline the extensions to the STG-machine implementation required for profiling. This section should be relevant even if the reader is not familiar with the details of the STG-machine (Peyton Jones [1992]) since we relegate the details of the STG-machine to Appendix A.

Mapping the abstract push-enter semantics onto the STG-machine description does not pose any particular problems. The main conceptual difference is that the STG-machine description is a "small-step" state transition system, while the abstract semantics are "big-step". The STG-machine state must record all information required for any remaining evaluation required by the corresponding big-step rule. For example, when a `case` is evaluated the alternatives must be recorded in the state. For the profiled implementations we have to save the enclosing cost centre if it is to be restored when evaluation returns.

Implementation of lexical scoping

The implementation of lexical scoping is quite straight forward. The main STG-level extensions required are:

- A *current cost centre* is added to the machine state. Any execution costs are attributed to the *current cost centre*.
- All heap closures have a cost centre attached to them. Whenever a closure is entered it loads its cost centre into the current cost centre, except for subsumed top-level functions.
- Evaluation of a `case` saves the enclosing cost centre with the continuation for the alternatives on the return stack. It is restored when the constructor is returned and the appropriate alternative evaluated.

A formal description of the STG-level manipulation of cost centres for lexical profiling is given in Appendix A.4.

Implementation of evaluation scoping

The implementation of evaluation scoping requires a second mechanism for saving and restoring cost centres. The STG-level modifications for evaluation scoping have a lot in common with the modifications required for lexical scoping. Below is a complete list of the STG-level modifications with the differences with lexical scoping *highlighted.*

- A current cost centre is added to the machine state. Any execution costs are attributed to the current cost centre.
- All heap closures have a cost centre attached to them. *This is loaded into the current cost centre when a thunk or constructor is entered. Entering a λ-abstraction or performing a partial application update (if there are not enough arguments available) does not load the closure's cost centre.*
- Evaluation of a `case` saves the enclosing cost centre with the continuation for the alternatives on the return stack. It is restored when the constructor is returned and the appropriate alternative evaluated.
- *The update mechanism is extended to save the enclosing cost centre in the update frame.* It is restored if the update is triggered by a partial application. If the update is triggered by a returning constructor the cost centre is not restored (see the Var_{Thunk} rule).
- *The update mechanism is also used to save/restore the enclosing cost centre when an* `scc` *expression is evaluated except that a dummy update frame is used.* This "update" does not actually update a closure, but just restores the cost centre if the "update" is triggered by a partial application. If the "update" is triggered by a returning constructor no action is taken. A simple optimisation can be performed which omits the dummy update if the result is known to be a data constructor.

Compiler analysis may determine that some unevaluated closures will only be evaluated once and omit the update (Launchbury et al. [1992]; Marlow [1993]). However, the demanding cost centre must still be restored when evaluation has completed. Unfortunately, we don't have an update frame to detect when this occurs. This problem is solved by pushing a dummy update frame, like that used for `scc` expressions, that restores the cost centre but does not actually perform an update.

We do not include a full description of the STG-level manipulation of this evaluation scoping semantics. It can be easily derived from the STG-level description of the hybrid profiling scheme in Appendix A.6 (see A.6.5).

Alternative implementation of evaluation scoping

Having extended the update mechanism to save/restore cost centres we observe that it is possible to make use of the update mechanism whenever a cost centre is saved or restored — removing the need for saving and restoring the cost centre when a `case` is evaluated. This can be done if the costs of entering *all* values, including constructors, are considered to be subsumed, i.e.

$$\text{SUB}'_e(z, as, cc_z, cc) = cc$$

Under this scheme a constructor is entered in the context of the enclosing cost centre cc, not the cost centre of the constructor cc_z. The (small) cost of entering a constructor and returning its value is attributed to the demanding cost centre. The cost centre enclosing an expression, evaluated in the context of a different cost centre, is *always* restored using the update mechanism by the Var_{Thunk} or SCC rule that entered the expression. This removes the need to restore the cost centre in the *Case* rule since the cost centre returned with the constructor cc_C is always the same as the cost centre cc that enclosed the `case` expression.

The only problem with this implementation is that the cost centre of the result is not available during an update. The returned cost centre cc_z is the same as the entered cost centre cc_e. Thus, the cost of performing the update is attributed to the cost centre of the closure being updated cc_e. Any copies of the closure constructed by the update mechanism will have the cost centre cc_e attached, unless an alternative mechanism for returning the cost centre to be attached to these closures is introduced.

Though this alternative evaluation implementation is not as clean, it does highlight the close relationship between evaluation scoping and the underlying lazy evaluation mechanism — hence the term *evaluation scoping*. Our initial evaluation scoping implementation used this semantics (extended with a return/update mechanism that ensured that the cost centre of the result is attached to closures constructed by the update mechanism). A

formal STG-level description of this alternative implementation of evaluation scoping is given in Appendix A.5.

Implementation of hybrid profiling scheme

The hybrid implementation requires both save/restore mechanisms. The complete list of STG-level modifications is given below. The differences with the standard implementation of evaluation scoping are *highlighted.*

- A current cost centre is added to the machine state. Any execution costs are attributed to the current cost centre.
- All heap closures have a cost centre attached to them. *Whenever a closure is entered it loads its cost centre into the current cost centre, except for subsumed top-level functions and λ-abstractions declared in the scope of a CAF or DICT cost centre.*
- Evaluation of a `case` saves the enclosing cost centre with the continuation for the alternatives on the return stack. It is restored when the constructor is returned and the appropriate alternative evaluated.
- The update mechanism is extended to save the enclosing cost centre in the update frame. It is restored if the update is triggered by a partial application *and the cost centre of the λ-abstraction being entered is a CAF or DICT cost centre.*
- A dummy update mechanism is introduced for `scc` expressions and single-entry closures. This "update" restores the cost centre if the "update" is triggered by a partial application *and the cost centre of the λ-abstraction being entered is a CAF or DICT cost centre.*

Unfortunately a λ-abstraction may not "know" its cost centre at compile time. Instead a runtime test has to be performed when a λ-abstraction is entered. To ensure this runtime test is as efficient as possible each cost centre contains a boolean flag indicating if it is a special "subsumed" cost centre. This is set for all CAF and DICT cost centres. Of course, if the enclosing cost centre is known at compile time, as it is for any λ-abstractions declared in the static scope of an `scc` annotation, the appropriate code can be generated at compile time and the test omitted.

A formal description of the STG-level manipulation of cost centres for our hybrid profiling scheme is given in Appendix A.6. Unfortunately this profiling scheme has not yet been implemented, though an implementation should be available with the next public release of the compiler.

5.6 Runtime System

Having described the manipulation of cost centres at the abstract machine level we now outline the main features of the low-level runtime implementation. In line with the discussion in Section 2.2.4 we have attempted to minimise the impact of the cost centre manipulation and runtime bookkeeping. In particular, we have ensured that all data requirements are declared statically and all inlined profiling instructions are simple assignments or counter increments (apart from the runtime test required by the hybrid profiling scheme).

A complete description of the implementation is beyond the scope of this thesis. Anyone interested in all the gory details should examine the (mostly documented) source code distributed with the Glasgow Haskell compiler. The following sections provide a brief overview of the main extensions to the runtime system required for profiling.

5.6.1 Flexible code generation

The Glasgow Haskell compiler generates C as its target code. Though we had to make significant modifications to the code generator for profiled execution we have attempted to be as general as we can in the code generated. Substantial use of C macros has been made to enable the bookkeeping performed at each profiling event to be easily modified.

We have also put considerable effort into generating C code that has a variable sized closure header (see Figure 5.13). This allows us to attach additional runtime information to every closure without modifying the code generator — we just have to modify the macro definitions. This feature is used by the cost centre profiling and other runtime system profiling. For example, the gathering of the closure lifetime and update age data presented in Sansom & Peyton Jones [1993] required a creation time field to be added to every closure.

	Fixed Code	Flexible Code
Space allocated for n word closure	`1 + n`	`1 + _HDR + n`
Indexing ith word of closure	`node[i]`	`node[_HDR+i]`

Initialise n word closure	`node[0] = hdr;`
	`node[1] = val1;`
	`...`
	`node[n] = valn;`
		`_INIT(node,hdr,n);`
		`node[_HDR+1] = val1;`
		`...`
		`node[_HDR+n] = valn;`

Figure 5.13: Example code generated for flexible closure layouts

5.6.2 Cost centres

During compilation we statically declare a *cost centre* structure for each `scc` annotation encountered in the module being compiled. Within a module, annotations with the same label refer to a single cost centre. However, annotations with the same label which reside in a different modules refer to different cost centres. These costs are currently reported separately, but could easily be combined in the profiling report.

Within each cost centre structure we store the following information:

- The **label** of the cost centre.
- The name of the **module** containing the `scc` annotation.
- The module **group** specified as a compiler option. If no group is specified the module name is used.
- For the hybrid profiling scheme we include a flag indicating if the cost centre is a special "subsumed" cost centre. This is set for CAF and DICT cost centres.[2]
- Any statistical meters we want to accumulate during execution. Currently we record:
  - `scc` enters,
  - the number of `scc` sub-expressions entered,
  - $\texttt{scc}_{caf}$ and sub-$\texttt{scc}_{caf}$ enters,
  - $\texttt{scc}_{sub}$ enters,[2]

[2]Not yet implemented.

- $\texttt{scc}_{dict}$ and sub-$\texttt{scc}_{dict}$ enters,[2]
- the number of thunks evaluated and closures allocated,
- the total heap space allocated, and
- the number of time ticks (see Section 5.6.6).

- Any temporary data needed during cost centre processing.

During execution a pointer to the cost centre structure is used to identify the cost centre. A special location, `_CCC`, is declared to store the current cost centre. This is initialised to the cost centre `MAIN` at the start of execution. Profiled events are attributed to the cost centre identified by `_CCC`.

5.6.3 Registering cost centres

When execution has completed we need to be able to access all the cost centres so we can produce a profile report. Ideally we would like to link all the cost centres together as they are declared during compilation. However, this is not possible with separate module compilation. So we are left with the task of *registering* each cost centre at runtime.

One solution is to register each cost centre the first time it is set to the current cost centre. Unfortunately this requires a conditional test *every* time an `scc` expression is executed.

An alternative solution, which we have adopted, is to traverse the module dependency graph at the start of execution, registering all the cost centres. In each module we declare a small routine that registers all the cost centres declared in that module and calls the registering routine of each imported module. At the start of execution, before evaluation of the program actually begins, the runtime system calls the registering routine for the `Main` module, and ensures that any prelude cost centres are registered. This ensures that the entire module structure is traversed and all the cost centres registered.

Care must be taken to ensure that any cycles in the module dependency graph are dealt with correctly. Each module is marked when its registering routine is first executed. Any subsequent calls to the registering routine observe that the module has been marked and simply return.

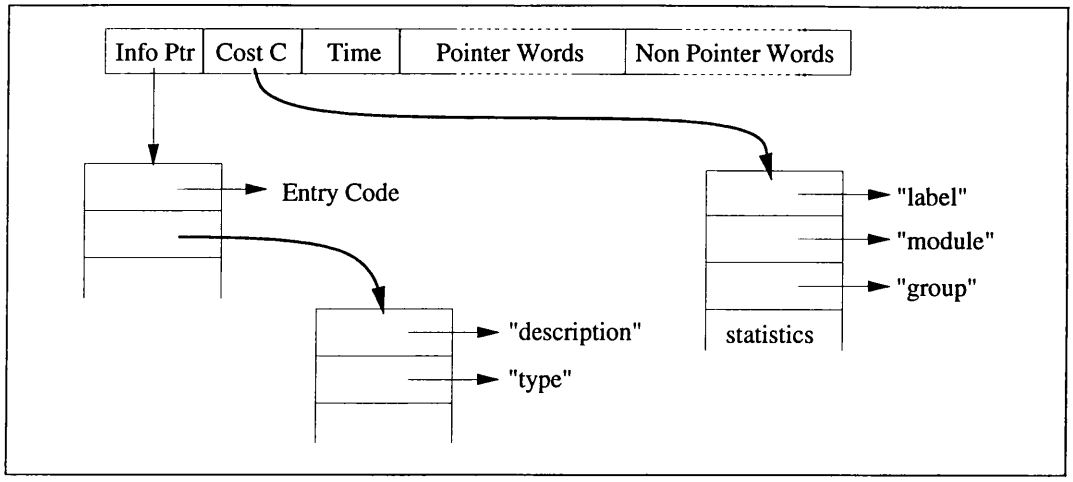


Figure 5.14: Closure layout

5.6.4 Closure layout

When profiling is enabled each closure has two words added to its header: a cost centre and a creation interval. The layout of each closure is shown in Figure 5.14. This is specific to the STG-machine. The first word of the closure is the *info pointer*. It points to the *info table*: a static structure containing information about the closure. In particular it contains the code to execute when the closure is entered. When profiling we add a pointer to some additional information describing the closure (see Section 5.6.5). The second word points to the cost centre responsible for building the closure and the third word the time interval during which the closure was created. Following this is a block of words containing pointers and a block containing non-pointers. The distinction between the two is for garbage collection purposes.

Thus, for every heap-allocated closure we can identify:

- Which "part" of the program created the closure, as indicated by the attached cost centre.
- What the closure is, as described by the additional information stored in the info-table (see Section 5.6.5).
- The time the closure was created.

This information is used when profiling the contents of the heap (see Section 6.3).

5.6.5 Closure descriptions

Each info table points to a closure description record that describes the closure.[3] This description consists of two strings: a type string and a description string derived from the original source. For example, an evaluated cons cell would have type `"List"` and description `":"`.

The type and description strings derived from the source are described below. Some examples are shown in Figure 5.15. Note that a particular closure's description changes when it is updated with its result.

Type strings

The type string is normally just the type constructor — any type parameters are not included in the type string. However, for function types we do examine the type parameters, rather than just using a type string `"->"`, producing a string of the form `"->>>`*resultty*`"` where there is one `>` for each argument type omitted and *resultty* is the type string of the final result type. Polymorphic types, which are not known at compile time, are given the unknown type string `"ty"`. Dictionary types introduced by the compiler are given the fixed type string `"dict"`.

We do not claim that these type strings are in any way ideal — we just chose a scheme that seemed reasonable. Any suggestions for improvements are welcomed.

Description strings

The description string depends on the form of the closure:

- **Thunks**, i.e. unevaluated closures, are described by the name of the declaration. However, if the declaration is anonymous or introduced by the compiler the description is derived from the expression on the right-hand-side using the outermost identifier being applied. If this is an application of a higher order argument the identifier is the argument name not the actual function being applied. The applied

[3] We introduce a separate description record, rather than including the description strings directly in the info table since the description record actually contains some additional data fields that cache runtime information such as hash values. These data fields cannot reside in the info table as this is declared constant and cannot be modified during execution.

Expression	Type	Description
`map = \ x y -> ...`	`"->>List"`	`"map"`
`result = map g xs`	`"List"`	`"result"`
... after evaluation	`"List"`	`":"`
`incall = map (+1)`	`"->List"`	`"incall"`
... after evaluation	`"PAP"`	`"PAP"`
`? = \ x y -> map x y`	`"->>List"`	`"\@map"`
`? = map g xs`	`"List"`	`"@map"`
`? = let ... in map g xs`	`"List"`	`"l@map"`
`? = case map g xs of ...`	need alts	`"c@map"`

Figure 5.15: Example type and description strings

identifier is prepended with an `@` if it is a normal application or a `#` if it is a primitive application. In addition a string summarising any `let` or `case` expressions enclosing the application (in the transformed program) is prepended.

- **Manifest functions** are also described by the name given to the declaration. If the declaration is anonymous or introduced by the compiler the name is derived from the right-hand-side, as it is for thunks, the only difference is a `\` is prepended.

- **Constructors**, i.e. evaluated data closures, are described by the actual data constructor used to build the data object. This is always known.

- **Partial applications** are built at runtime. Since there is only one info-table for partial applications we have to use a single description `"PAP"`. This is also used for the partial application type string. The runtime system could construct a more meaningful description using the description of the function being applied, but we do not do this.

Again, we do not claim that these description strings are in any way ideal. It may be necessary to dump the intermediate code (see Appendix B.1) to determine exactly what expression a particular description refers to. We will move to improve them if user feedback indicates it would be worthwhile. Possible improvements include:

- Giving a binding introduced by the compiler a name derived from the enclosing source level declaration.
- Including the module name and line number in the description string.

5.6.6 Time profiling

Underlying the time profiles produced by our profiling system (see Chapter 6) is a very simple execution sampling mechanism implemented with Unix signals. We interrupt the execution of the program every 20ms using the `setitimer` system call. A customised routine to handle each interrupt is specified using the `signal` system call. During each interrupt the interrupt handler increments the `time_tick` counter for the cost centre currently referenced by the current cost centre (`_CCC->time_tick`). As the value of `_CCC` is required by the interrupt handling routine it must be stored in a memory location — it is not possible to optimise it into a real machine register.

5.6.7 Heap profiling

If the user requests a heap profile, execution is suspended at regular intervals (specified by the user), and the entire heap garbage collected. During this garbage collection a profiling routine is called once for each live closure in the heap. It is passed a pointer to the closure and the closure's size. From this it can extract the closure's cost centre and description, and incrementally build up the data required for the requested heap profile (see Section 6.3). When the garbage collection is completed a profiling finalisation routine is called and the data for this heap sample is written to a file.

Interrupting execution

The heap profile interrupt will occur at some arbitrary point during the execution. Unfortunately the garbage collector can only be invoked when the heap is in a consistent state and all pointers into the heap are known. The only time this is guaranteed to be the case is when a heap overflow check has failed and the special code which tidies up the state and makes the roots known to the garbage collector has been executed.

This problem is solved by introducing a global flag `interval_expired` that indicates when a heap profiling interval has expired. Each heap overflow check is modified to test

the availability of heap space and the `interval_expired` flag:

```
if (Hp + HpRequest > HpLim || interval_expired) {
    ... tidy up and invoke garbage collection ...
}
```

The `interval_expired` flag is set by the interrupt handler when the heap profiling interval expires and normal execution is resumed. The required garbage collection is then invoked at the next heap allocation.

Profiled garbage collection costs

The garbage collection overheads of profiled execution are vastly different to the garbage collection overheads of normal execution.

- Garbage collection is performed whenever a heap sample is required.
- Each garbage collection includes the additional cost of collating the required profiling data.
- We currently use a two-space copying garbage collector, rather than the generational garbage collector used during normal execution.

As far as time profiling is concerned we ignore the profiled garbage collection costs by disabled the execution sampling during garbage collection. (Though we do plan to provide a brief summary of the garbage collection costs with the cost centre profile (Section 6.2) in the next release.)

If required, detailed information about the profiled garbage collection overheads can still be obtained by requesting the garbage collection statistics using the `-s` or `-S` runtime option. More importantly, information about the normal garbage collection overheads can be obtained by requesting the garbage collection statistics during a normal execution.

5.7 Profiling Overheads

One important consideration of any profiling system is the overheads it imposes on execution time and space requirements. If the profiling overheads are too high the tool may become unusable. Our profiling system imposes a number of overheads:

	Optimised C Execution			Portable C
	Normal	Profiled	Overheads	Profiled
Useful Reduction Time	2,709s	4,437s	64%	8,576s
GC and Heap Profiling Pauses	258s	2,478s	n.a.	5,214s
Total Execution Time	2,967s	6,915s	133%	13,790s
Total Heap Allocation (Mb)	12,119	19,873	64%	19,873
Maximum Heap Residency (Mb) (compiling `TcExpr.lhs`)	3.5	5.7	63%	5.6
Executable Size (Mb)	6.7	9.7	44%	14.1

Figure 5.16: Profiling Overheads Compiling the Compiler

- The manipulation of cost centres and time sampling interrupts (every 20ms) reduce execution speed somewhat.
- Heap profiling increases the garbage collection overheads (see Section 5.6.7).
- The heap space occupied and allocated is increased as every closure has two extra words storing the cost centre and creation interval.[4] (This space overhead is discounted in the allocation/live heap data reported in the profiling output.)
- Executable size increases, due to cost centre manipulation code and static profiling data structures and strings.
- The current profiling implementation uses the two-space copying garbage collector. This imposes an additional 100% heap space overhead for the second semi-space.

Figure 5.16 provides a summary of the optimised profiling overheads measured over the compilation of the whole compiler. The profiled execution times are for lexical profiling runs generating both a cost centre and a heap profile with a heap sampling interval of one second and time sample of 20ms. Profiled execution imposes a basic 64% execution time overhead. Additional pauses to profile the heap (every second) brought the total execution overhead to 130%. This additional overhead is only incurred if a heap profile is requested. We consider this to be an acceptable execution overhead.

[4] It is possible to build a version of the profiler that does not store the creation interval in each closure, removing the ability to produce a heap profile broken down by creation time. This would halve the 64% space overhead.

The space overhead is about 64%. This applies to the amount of heap allocated and the maximum heap residency. Though this seems quite acceptable, it excludes the additional space required by the two-space copying garbage collector. This brings the total heap space overhead to more than 200%. This is not acceptable, especially since we expect to be profiling programs with space problems. To address this problem we are planning to develop an implementation of the profiling runtime system that uses our generational garbage collector (Sansom & Peyton Jones [1993]). This does not require the additional semi-space since the major collection uses an inplace compaction algorithm. Data for the heap profiles will be gathered using the mark phase of the major collection.

All the profiles presented in Chapters 6 and 7 are produced by a version of the profiler that compiled programs using portable C compilation. Using a more sophisticated code generation route (as we now do) improves the performance by a factor of more than 2, for both profiled and unprofiled programs. Timings for portable C profiled execution are given in Figure 5.16 for comparison.

Slower profiled execution does not cause us any problems because we are interested in identifying relative costs and quantifying relative improvements. In fact, slower profiled execution may even have a benefit, since it results in an increased number of timer samples, improving the accuracy of the resulting profile; i.e. if your profile does not contain enough timer samples run it on a slower machine! A more desirable solution would be to increase the sampling frequency, but not all Unix machines support this.

5.8 Correctness

We have no formal proof that the costs attributed to an annotated source expression are a true reflection of the costs that should be attributed to that expression for the profiling semantics being used.

However, we have attempted to be rigorous in our specification and implementation of the profiling system. The main components of this rigour are the abstract cost semantics (Section 4.2), the push-enter cost semantics (Section 5.5) and the STG-machine operational semantics (Appendix A). These provide us with very useful formalisms at different levels of abstraction. Though we have no formal proof that the *current cost centre* identified by the STG-level operational semantics actually corresponds to the cost centre

context identified by the abstract cost semantics, the formalisms provide us with a strong notion that this is indeed the case.

At a more practical level, using the profiler has provided us with a considerable amount of evidence that supports the accuracy of the profiling results reported. When examining the profile of a program we consider the question:

Can we explain the (often surprising) profiling results?

If we can convince ourselves that the profiling results are plausible then all is well (we hope). If not, we attempt to identify the reason for the inconsistency: Have we overlooked an explanation or is there a bug in the implementation of the profiler or compiler? A very useful step is to obtain a dump of the optimised code from within the compiler. This optimised dump can be of benefit to anyone attempting to understand their profile since it contains the optimised code that is actually being profiled. Any inconsistencies are usually explained by the (sometimes erroneous) transformations which were performed within the compiler.

Chapter 6

Profiling Output

The current implementation of the profiler produces a number of different profiling outputs:

- An aggregate cost centre profile.
- A serial heap profile.
- A serial time profile.

Each of these profiles is described in the following sections. We present example profiles and compare the profiles produced by lexical scoping with those produced by evaluation scoping. The relevant **ghc** user documentation can be found in Appendix B.

6.1 Example program: `clausify`

The example profiles that are presented in this Chapter are generated by a Haskell version of the program `clausify` (Runciman & Wakeling [1993]). This program was chosen as it has already been the subject of extensive profiling and improvement by Runciman and Wakeling using the **hbc/lml** heap profiler. We were interested in discovering if our profiling output revealed anything about the execution of the program that the **hbc/lml** heap profiler did not. We profile the final version of the program developed in Runciman & Wakeling [1993].

Before describing the profiles we give a brief summary of the `clausify` program and describe the profiling annotations used. (The summary has been taken from Runciman &

Wakeling [1993].)

`Clausify` takes as input a series of propositional formulae, and produces their clausal form equivalents. The required transformation of each proposition to a set of clauses can be specified by the following rules:

- *elim* eliminates equivalence and implications:

$$\begin{aligned} p = q &\rightarrow (p \Rightarrow q) \wedge (q \Rightarrow p) \\ p \Rightarrow q &\rightarrow \neg p \vee q \end{aligned}$$

- *negin* makes negations the innermost connectives:

$$\begin{aligned} \neg\neg p &\rightarrow p \\ \neg(p \vee q) &\rightarrow \neg p \wedge \neg q \\ \neg(p \wedge q) &\rightarrow \neg p \vee \neg q \end{aligned}$$

- *disin* pushes disjuncts within conjuncts:

$$\begin{aligned} p \vee (q \wedge r) &\rightarrow (p \vee q) \wedge (p \vee r) \\ (p \wedge q) \vee r &\rightarrow (p \vee r) \wedge (q \vee r) \end{aligned}$$

- *split* splits up the conjuncts:

$$\begin{aligned} p \wedge q &\rightarrow p \\ &\qquad\ q \end{aligned}$$

- *unicl* forms a set of unique non-tautologous clauses:

$$p_1 \vee \ldots \vee p_n \vee \neg q_1 \vee \ldots \vee \neg q_m \rightarrow (\{p_1, \ldots, p_n\}, \{q_1, \ldots, q_m\})$$

  A clause (ps, qs) is tautologous if $(ps \cap qs) \neq \emptyset$.

The implementation of the transformation rules uses the following data definition to represent propositional formulae:

```
data Formula = Sym Char
             | Not Formula
             | Dis Formula Formula
             | Con Formula Formula
             | Imp Formula Formula
             | Eqv Formula Formula
```

At the heart of the program is a "pipeline" composition of several functions, each corresponding to one of the rules above.

```
clauses = unicl . split . disin . negin . elim
```

Appendix C contains the complete Haskell source for the final version of `clausify` developed by Runciman & Wakeling [1993].

For the purposes of profiling we use the same input as Runciman & Wakeling [1993] so that our profiling results are comparable. Namely the proposition:

$$(a = a = a) \;=\; (a = a = a) \;=\; (a = a = a)$$

Though this reduces to the single clause, $(\{a\}, \emptyset)$, it generates a substantial amount of work as the intermediate formulae are extensive.

Cost centre annotations

To profile `clausify` we first have to identify the expressions of interest. Initially we added explicit `scc` annotations to each element of the pipeline:

```
clauses = (\x -> scc "unicl" unicl x) .
          (\x -> scc "split" split x) .
          (\x -> scc "disin" disin x) .
          (\x -> scc "negin" negin x) .
          (\x -> scc "elim"  elim x)
```

We had to introduced the `\x`'s to expose the application sites to the `scc` annotations when using evaluation scoping. If we were only interested in using lexical profiling, the following, less intrusive, annotations would have sufficed:

```
clauses = (scc "unicl" unicl) . (scc "split" split) .
          (scc "disin" disin) . (scc "negin" negin) .
          (scc "elim"  elim)
```

We also give an example profile generated using the automatic annotation scheme provided by the compiler.

6.2 Cost Centre Profile

The **cost centre profile** shows the proportion of execution time and heap allocation attributed to each cost centre during a run of the program. It is generated with the `-p` or

-P runtime option (see Appendix B.2). The output consists of a simple text file displaying formatted data. Example cost centre profiles are presented in Figures 6.1, 6.2.

The profile displays the date the program was run, the command used to generate the profile, the total execution time (measured in seconds), the total number of time ticks and the tick interval (normally 20ms), the total heap allocation (measured in bytes) and the total number of closures allocated. For each cost centre, introduced with an explicit source annotation or by the compiler, the -p profile reports:

COST CENTRE: The cost centre label. The cost centre module and group are also reported, but they have been omitted from the example profiles to ease presentation.

scc: The number of instances of the scc annotated expression that were evaluated. If the entire body of a function is annotated with an scc expression the scc entry count is the number of function calls.

subcc: The number of scc annotated sub-expressions that were evaluated. The costs of these sub-expressions are attributed to their cost centre, not this cost centre. Unfortunately, this count does not identify which cost centres these sub-costs are being attributed to. This would require counts to be associated with cost-centre pairs (see Section 8.4.2).

%time: The proportion of CPU time spent evaluating instances of the annotated expression. (The *current cost centre* is sampled every 20ms.)

%alloc: The proportion of the total heap allocation that was allocated by the evaluation of the instances of the annotated expression. (The space allocation for each closure is attributed to the cost centre stored in the closure.)

A more detailed profiling output can be requested using the -P runtime option. This also reports:

cafcc: This consists of two counts concerning the evaluation of CAFs. The first is the $\mathtt{scc}_{caf}$ entry count for this cost centre. The second is the number of unevaluated CAFs whose value was demanded by this cost centre.

thunks: The number of thunks allocated by this cost centre that were subsequently evaluated.

`closures`: The number of closures allocated by this cost centre.

`ticks`: The number of time ticks attributed to this cost centre. This is used to calculate the `%time`.[1]

`bytes`: The number of bytes allocated by this cost centre. This is used to calculate the `%alloc`.[1]

Sorting the profile

The profile can be sorted by `%time`, `%alloc`, or alphabetically by module and label. The example profiles are sorted by `%time`, with all CAF cost centres placed at the end. This is the default sorting. It places all the "expensive" cost centres at the top of the profile.

Special cost centres

There are a number of special cost centres that can be seen in the example profiles. The cost centre `"MAIN"` is the initial cost centre, set at the start of execution. It is attributed with the costs of processing the I/O requests, and constructing the responses. In particular, the costs of actually performing the I/O (reading and writing the characters) is attributed to `"MAIN"`. It also subsumes any costs of evaluating `main` that are not attributed to some source cost centre.

`"Prelude:CAF"` is attributed with the costs of evaluating all the CAFs in the prelude. These CAFs may have been introduced by the compiler when compiling the prelude.

`"Prelude:DATA"` is attributed with the costs of all static data closures. These consist of single arity constructors, all the characters, and some small integers. This cost centre is never attributed any costs by our evaluation profiler because our evaluation implementation does not load the current cost centre on entry to a data closure.

6.2.1 Lexical vs. Evaluation cost centre profiles

We now compare a lexical profile and an evaluation profile of `clausify`, generated using the explicit `scc` annotations described in Section 6.1. This highlights problems with both

[1]The `ticks` and `bytes` data fields have been omitted from the example cost centre profiles to ease presentation.

```
    Tue Apr 26 18:00 1994 Time and Allocation Profiling Report
                (Lexical Scoping)

       clausify-lex +RTS -P -RTS

     total time  =        4.04 secs   (202 ticks @ 20 ms)
     total alloc =   3,162,380 bytes  (197701 closures)

COST CENTRE        scc subcc  %time %alloc   cafcc    thunks closures
disin                1     0    8.9    6.3   0 0     12362    12362
split                1     0    2.0    3.4   0 0      5345     5346
negin                1     0    0.5    0.1   0 0       198      294
elim                 1     0    0.0    0.2   0 0       198      330
MAIN                 1     0    0.0    0.1   0 8        34      129
unicl                1     0    0.0    0.0   0 1         0        0
CAF:unicl            0     0   78.2   89.7   1 2    115497   178597
CAF:d.Eq.c10         0     0    4.0    0.0   1 0         2        6
Prelude:CAF          0     0    3.0    0.1   9 2       138      278
CAF:main             0     5    2.0    0.2   1 4       198      330
CAF:elem.c82         0     0    1.0    0.0   1 0         1        3
Prelude:DATA         0     0    0.5    0.0   0 0         0        0
CAF:stg              0     0    0.0    0.0   3 0        18       24
CAF:spaces           0     0    0.0    0.0   1 0         0        2
```

Figure 6.1: Lexical Scoping Cost Centre Profile (explicit annotation)

```
    Tue Apr 26 18:01 1994 Time and Allocation Profiling Report
                (Evaluation Scoping)

       clausify-eval +RTS -P -RTS

     total time  =        3.52 secs   (176 ticks @ 20 ms)
     total alloc =   3,162,380 bytes  (197701 closures)

COST CENTRE        scc subcc  %time %alloc   cafcc    thunks closures
unicl                1     0   91.5   89.7   0 2    115490   178584
disin                1     0    5.1    6.3   0 0     12362    12362
split                1     0    3.4    3.4   0 0      5345     5346
MAIN                 1     5    0.0    0.3   0 13      319      654
elim                 1     0    0.0    0.2   0 0       198      330
negin                1     0    0.0    0.1   0 0       198      294
Prelude:CAF          0     0    0.0    0.0   9 1        51       80
CAF:stg              0     0    0.0    0.0   3 0        18       24
CAF:unicl            0     0    0.0    0.0   1 1         7       13
CAF:d.Eq.c10         0     0    0.0    0.0   1 0         2        6
CAF:main             0     0    0.0    0.0   1 0         0        3
CAF:elem.c82         0     0    0.0    0.0   1 0         1        3
CAF:spaces           0     0    0.0    0.0   1 0         0        2
```

Figure 6.2: Evaluation Scoping Cost Centre Profile (explicit annotation)

profiling schemes, thus revealing the need for the hybrid profiling scheme proposed in Section 4.4.3.

Origins of the hybrid profiler

The lexical profile is shown in Figure 6.1 and the evaluation profile in Figure 6.2. The most notable feature of the lexical profile is that 78% of the time is attributed to `"CAF:unicl"`. Examining the source of `clausify` (Appendix C) reveals that `unicl` is declared as a CAF that has a function result:

```
unicl = filterset (not . tautclause) . map clause
```

The lexical cost attribution attributes the costs of applying this function to the CAF cost centre introduced by the compiler (Section 4.4.1). Our explicit annotation of the reference to `unicl` is attributed with zero cost. In a similar way the costs of applying the method function embedded in the `Eq` dictionary are attributed to the cost centre `"CAF:d.Eq.c10"`[2] that constructed the dictionary. It is not immediately clear which part of the program is responsible for incurring these costs. In a large program, determining which part is responsible for these CAF costs can be a very intractable problem.

In the evaluation profile the CAF cost centres are only attributed with the small, one-off evaluation costs. The costs of applying the functions embedded in the CAF results are attributed to their application site. It turns out that all these CAF costs are attributed to `"unicl"`. However, if there were many different applications of the CAF results the application costs would be distributed between the cost centres of the different application sites.

The evaluation profile is much easier to interpret. However, it was only generated with very carefully placed cost centre annotations (Section 6.1). Just annotating the references, without exposing the application site, results in zero cost being attributed to each cost centre — all the costs are attributed to `"MAIN"`.

The hybrid profiling scheme should be a significant improvement since it will give us the best of both worlds: the simple annotation of lexical scoping, with a usable attribution of CAF and dictionary costs.

[2] Dictionary cost centre annotations, as described in Section 4.4.2, have not yet been implemented.

Timing differences

The different cost centre manipulation required by the two profiling schemes results in slightly different timings. The lexical profiler saves/restores the current cost centre whenever a `case` is executed where as our evaluation implementation only saves/restores the current cost centre when an unevaluated closure is entered. As execution of a `case` occurs more frequently the overhead of the lexical profiler is larger (observe the larger `total time` in the lexical profile).

Since the evaluation profiler does not load the cost centre when a data closure is entered, the cost of returning the value is attributed to the demanding closure. This can be observed in the zero costs attributed to `"Prelude:DATA"` and the smaller cost attributed to `"disin"`. Instead, these costs are attributed to `"unicl"`, the cost centre demanding the values.

6.2.2 Automatic annotation

Figure 6.3 shows a lexical profile of `clausify` generated by directing the compiler to annotate all top-level declarations (`-auto-all` compiler option). This produces a profile with a very different flavour:

- The profile identifies the top-level functions that are consuming all the execution time. These can then be examined for possible improvements.
- The `scc` counts report the number of times each top-level function was applied, revealing information about the structure of the execution.

This information can be used to direct low-level coding and data-structure improvements.

Comparing the `total time` in Figure 6.3 with Figure 6.1 reveals that profiled execution of `clausify` with automatic annotation is 10% slower than profiled execution with explicit annotation. There are two factors which contribute to this slower execution:

- The number of `scc` annotations results in more optimisations being curtailed (see Section 5.4.2).
- The bookkeeping overhead is larger since a lot more `scc` annotations are executed (observe the large `scc` counts in Figure 6.3).

```
    Tue Apr 26 18:02 1994 Time and Allocation Profiling Report
                (Lexical Scoping)

       clausify-lex-auto +RTS -P -RTS

     total time  =        4.44 secs   (222 ticks @ 20 ms)
     total alloc =   3,162,604 bytes  (197714 closures)

COST CENTRE          scc subcc  %time %alloc  cafcc    thunks closures
clause              5346 52398   37.8   76.1   0 0     99450   151850
insert             52398     0   20.7    0.0   0 0         0        0
disin'             12214 12164    9.0    6.2   0 0     12164    12164
tautclause          5346     0    7.2    7.4   0 1      5346    16037
split                  1     0    3.6    3.4   0 0      5345     5346
parse'                40    55    3.2    0.0   0 1         9       61
filterset'          5347     0    1.8    0.0   0 0         1        2
elim                 199   198    1.4    0.2   0 0       198      330
negin                231   230    0.9    0.1   0 0       198      294
clausify               1     4    0.0    0.1   0 1       160      208
disin                199   248    0.0    0.1   0 0       198      198
MAIN                   1     0    0.0    0.1   0 8        34      129
disp                   1     5    0.0    0.0   0 0        17       24
red                    8     0    0.0    0.0   0 0         0       16
while                 12     8    0.0    0.0   0 0         8        8
interleave             9     0    0.0    0.0   0 1         6        6
filterset              1     0    0.0    0.0   0 0         0        0
opri                  26     0    0.0    0.0   0 0         0        0
parse                  1     1    0.0    0.0   0 0         0        0
spri                  20    18    0.0    0.0   0 0         0        0
CAF:unicl              0  5346    5.0    6.1   1 2     10694    10697
Prelude:DATA           0     0    2.7    0.0   0 0         0        0
CAF:elem.c15           0     0    2.3    0.0   1 0         1        3
CAF:filterset          0  5348    1.8    0.0   1 1         7       16
Prelude:CAF            0     0    1.4    0.1   9 2       138      278
CAF:d.Eq.c25           0     0    1.4    0.0   1 0         2        6
CAF:clausifyline       0     2    0.0    0.0   1 1        12       17
CAF:main               0     1    0.0    0.0   1 0         3        8
CAF:redstar            0    24    0.0    0.0   1 0         0        6
CAF:clauses            0     4    0.0    0.0   1 1         4        6
CAF:spaces             0     0    0.0    0.0   1 0         0        2
CAF:tautclause         0  5346    0.0    0.0   1 0         0        2
```

Figure 6.3: Lexical Scoping Cost Centre Profile (`-auto-all` annotation)

6.2.3 Allocation rate

One general property that the cost centre profiles have revealed is that the rate of allocation is not constant across the different parts of the program — the %time and %alloc figures often differ by a considerable amount (observe `"insert"` in Figure 6.3). This is especially true in a heavily optimised implementation since the affect of the optimisations which reduce allocation is not uniform.

6.3 Heap Profiles

The **serial heap profile** shows how the amount of heap space (measured in bytes) required by the program varies over the execution of the program (measured in seconds). A graphical post-processor, `hp2ps` (see Appendix B.3.2), is used to convert the raw data gathered during execution into a PostScript[3] graph. Shaded bands provide further information about the contents of the heap (Section 6.3.1). The various heap profiling options are described in Appendix B.2.

An example heap profile of `clausify` is shown in Figure 6.4.[4] This profile has the heap contents broken down by the cost centre attached to each closure (`-hC` runtime option). The order of the key is the same as the order of the bands. This removes any ambiguity arising from the limited number of different shades that are available. The title displays the command used to generate the profile, a measure of the total cost of the program (the total area below the graph), and the date the program was run. The total cost of the `clausify` program is approximately 27Kbs (Kbyte-seconds).

6.3.1 Heap contents

There are a number of different criteria on which the contents of the heap can be broken down, based on the information attached to each closure (Section 5.6.4). These fall into three categories:

Producer profiles (Figure 6.4)

[3] PostScript is a registered trademark of Adobe Systems Incorporated.

[4] Figure 6.4 is equivalent to Figure 14 in Runciman & Wakeling [1993]. **ghc** does not have a mechanism equivalent to the subsequent improvement made to the **hbc/lml** compiler, which eliminated the lazy pattern matching space leak (Sparud [1993]; Wadler [1987]).

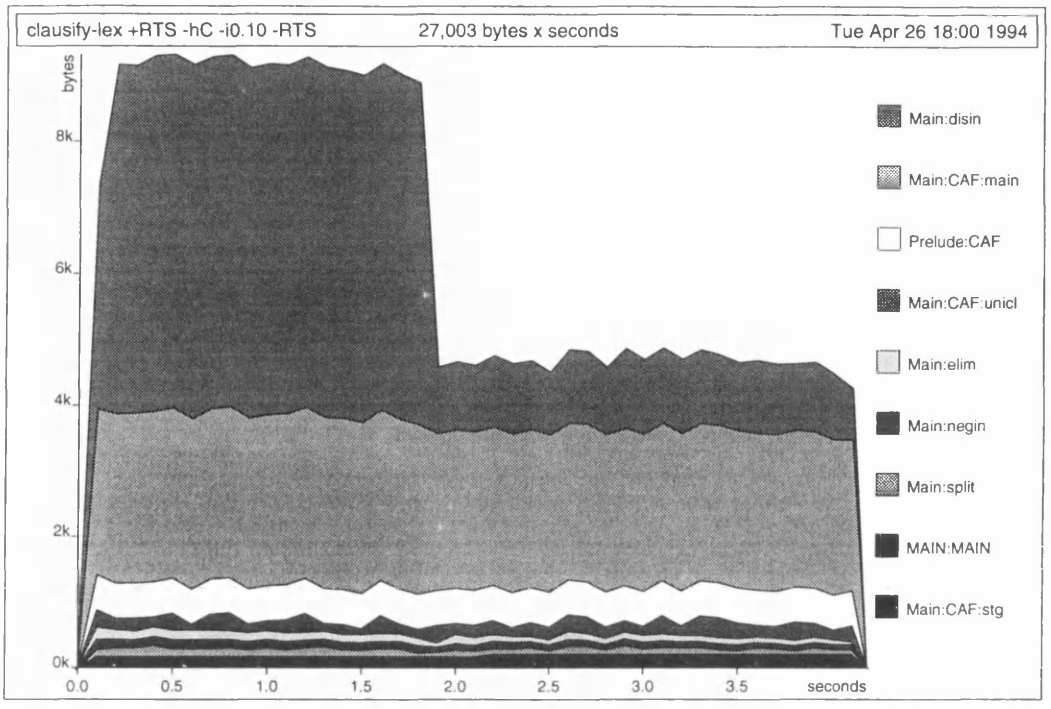


Figure 6.4: Heap Profile by Cost Centre (lexical scoping)

Producer profiles break down the contents of the heap by the cost centre attached to each closure. This can be done on a cost centre, module or group basis. They reveal which "part" of the program was responsible for producing the closures in the heap.

Description profiles (Figure 6.5)

Description profiles break down the contents of the heap by the static description string or type string (Section 5.6.5) attached to each closure. This reveals "what" closures make up the contents of the heap.

Creation time profile (Figure 6.6)

The creation time profile breaks down the heap by the time interval during which each closure was allocated. Each band shows the life of the closures created during a particular interval.

It is important that all bands are plotted in the profile and that the bands are stacked in the creation interval order, with the early intervals on the bottom. This required us to extend the `hp2ps` post-processor, adding the `-m`, `-i` and `-t` options (see Appendix B.3.2).

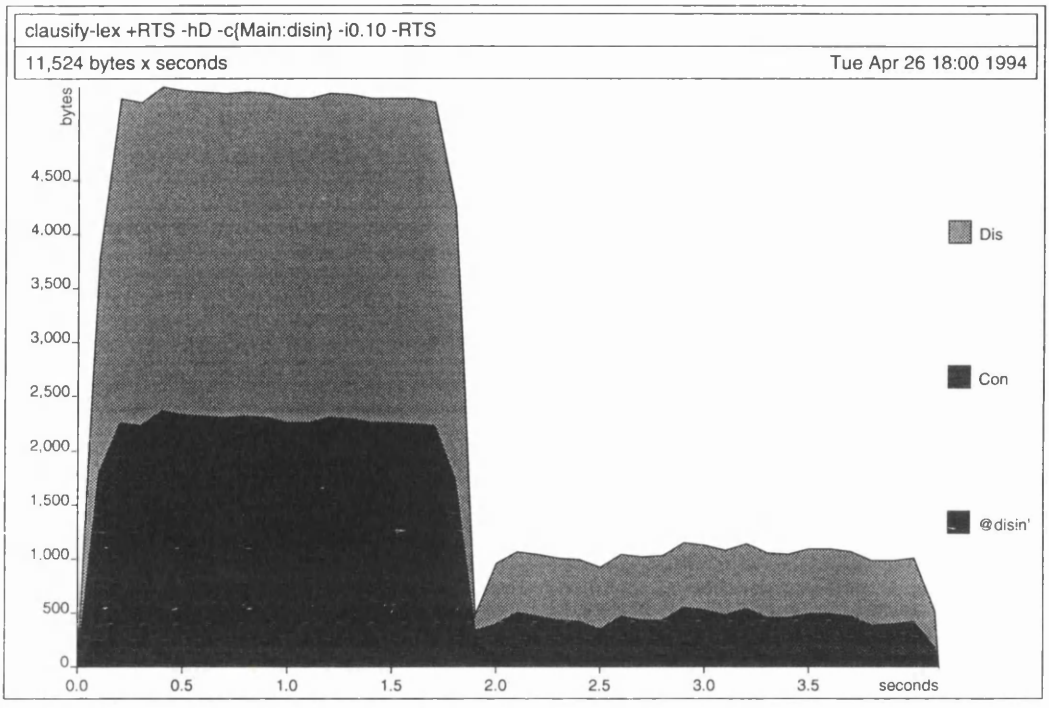


Figure 6.5: Heap Profile of "`disin`" by Description (lexical scoping)

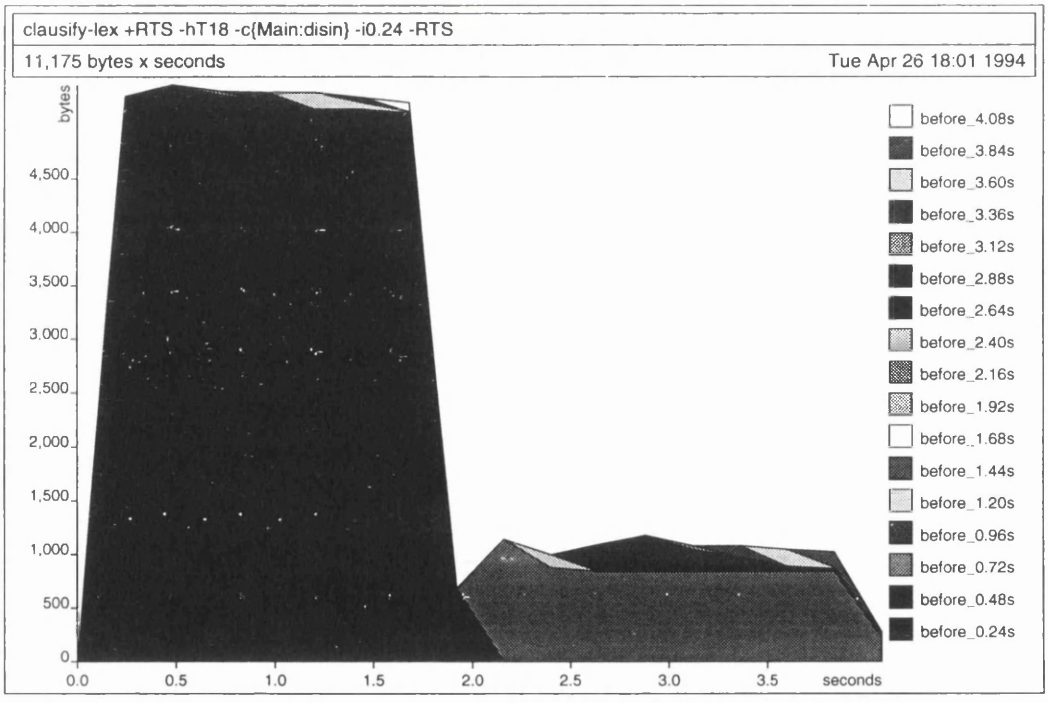


Figure 6.6: Heap Profile of "`disin`" by Creation Time (lexical scoping)

6.3.2 Heap selection

By default all live closures in the heap are reported in the heap profile, but the profile can be limited to a particular subset of closures. This allows the user to "focus" the profile on any problematic heap closures. Closures can be selected on the attached cost centre (label, module or group), description string, type string, closure kind, and closure age.

These selection features can be used to produce quite specific profiles that examine the behaviour of particular groups of closures. For example:

`-hD -c{Main:disin}` produces a heap profile, broken down by the description string, of the closures allocated by the cost centre "`disin`" (Figure 6.5)

`-hT -c{Main:disin}` produces a heap profile, broken down by the creation time, of the closures allocated by the cost centre "`disin`" (Figure 6.6).

A very good demonstration of the use of such heap profiles to investigate space problems can be found in Runciman & Wakeling [1993].

6.3.3 Comparison with other heap profilers

Our heap profiler is very similar to the heap profilers provided by the Chalmers **hbc/lml** compiler (Runciman & Wakeling [1993]) and the **nhc** compiler (Röjemo [1994]). Indeed, the same post-processor (`hp2ps`) is used to provide the graphical visualisation.

All these heap profilers provide producer and description profiles. However, there is an important difference between our producer profiles, which are based on the attached cost centre, and the **hbc/lml** and **nhc** producer profiles, which are based on static function, module, and group information. Cost centres enable the closures produced by an unprofiled function to be subsumed by the referencing function. This cannot be achieved by the other heap profilers. For example, examination of Figure 2 in Runciman & Wakeling [1992] reveals a large "band" attributed to `lib` (the standard prelude) rather than to the user's program source. With our profiler the costs of evaluating a prelude function are subsumed by the reference site. Consequently any closures constructed by the prelude function are attributed to the cost centre that referenced the function.

Using the automatic annotation of all top-level declarations (`-auto-all`) makes our producer profiles almost identical to the **hbc/lml** and **nhc** producer profiles. The only

difference is the subsuming of prelude functions. Perhaps we should provide a special version of the prelude, compiled with `-auto-all`, so that these functions can be identified if required.

The **nhc** profiler also provides *lifetime* and *retainer* profiles (Section 3.3.2). Our creation time profile provides similar information about the age of closures to the lifetime profile, though the semantics of the profiles are quite different. The lifetime profile reports the age each closure, currently residing in the heap, will live to, while the creation time profile shows the pattern of survival for the closures created in each interval. We have not developed an equivalent of the retainer profile, though this does look a promising direction for identifying the cause, rather than the presence, of space leaks (Runciman & Röjemo [1994]).

6.4 Serial Time Profile

We have also been experimenting with a **serial time profile**. The idea is to provide the programmer with a visual picture displaying the order of evaluation. Execution is divided up into a number of intervals and the time attributed to each cost centre during each interval is displayed. We currently use the same graphical post-processor as we do for the heap profiles, plotting bands showing the proportion of time (measured in ticks) vs. execution time (measured in seconds). An example serial time profile of `clausify` is shown in Figure 6.7.

Though this profile is an intriguing idea, it only provides a very rough picture of the execution behaviour. It suffers from a number of problems:

- The sampling process may miss small, but important, spurts of execution.
- The presentation is not very satisfactory since time is not always distributed in neat bands. (Though we have observed pipelines exhibiting bands of execution.)
- The intervals have to be large to provide a reasonable number of samples for each interval. However, a *moving average*[5] could be used to overcome this problem.

A more satisfactory profile might be provided by reporting the distribution of some fundamental execution event, such as closure entry, for each time interval. This would not

[5]The moving average is the average of the last n data points.

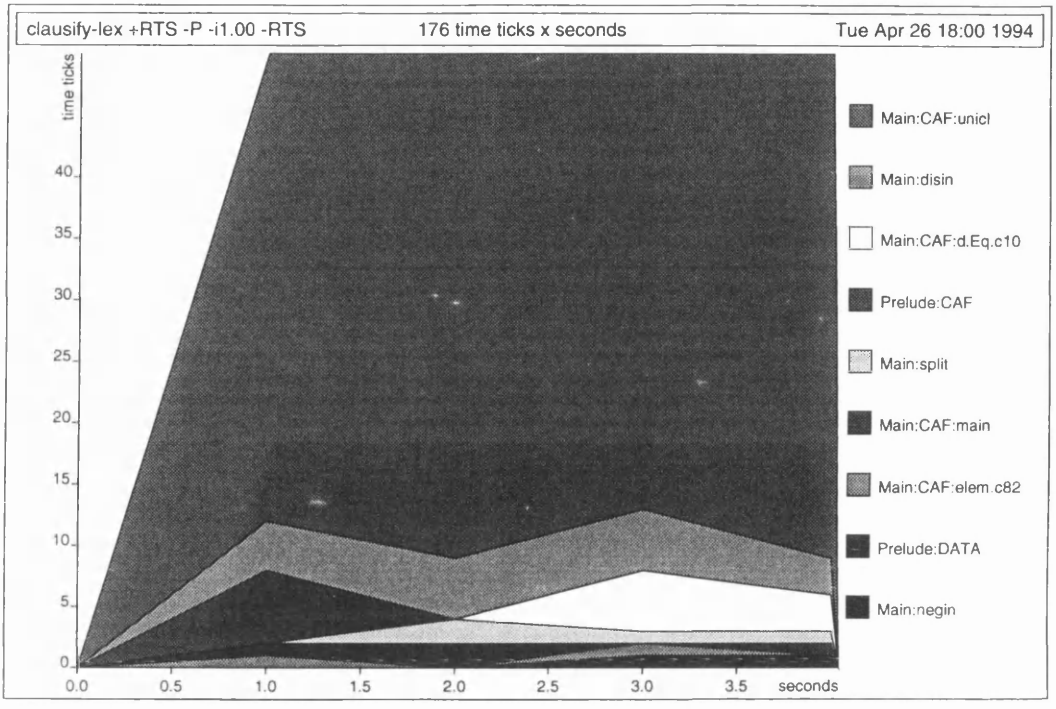


Figure 6.7: Serial Time Profile (lexical scoping)

suffer from the sampling problems mentioned above. Indeed, we could measure the "time" for the cost centre profile (Section 6.2) using closure entry counts (currently we only report thunk entry counts and closures allocated). The disadvantage with this approach is that the "time" spent in system routines is not observed because these routines don't enter closures.

6.5 `Clausify` Revisited

So, what have the **ghc** profiles revealed about `clausify`?

The time profiles revealed that `unicl` consumes about 90% of the execution time. This was not revealed by any of the heap profiles here or in Runciman & Wakeling [1993]. The `-auto-all` profile (Figure 6.3) revealed that this time is spent in `clause`, `insert`, and `tautclause`, throwing away all the clauses and duplicate a's.

The ability to focus directly on this hot-spot quickly led to a significant performance gain. Observing that all this "throwing away" relies on comparing characters we introduced unboxed characters (Peyton Jones & Launchbury [1991]) into the `Formulae` and

literal lists, recoding the functions that processed these characters. The source modifications can be found in Appendix C.2. These modifications resulted in a performance improvement of more than 25%, with less than an hour's work.

The creation time profile of "`disin`" reveals that all the `Dis` and `Con` closures produced by "`disin`" are created in two bursts. One at the start of execution, and one half way through. This behaviour is also revealed by Runciman & Röjemo [1994] using the **nhc** lifetime profiles. They go on to develop a further improvement after examining the **nhc** retainer profiles.

Chapter 7

Practical Applications

We have placed considerable emphasis on developing a practical profiler, suitable for profiling large applications. This Chapter demonstrates the use of the profiler for profiling large applications with detailed results from profiling the compiler itself (Section 5.1). We also report on the experiences some other users have had using the profiler (Section 7.2), before drawing some conclusions about the practical use of the profiler (Section 7.3).

7.1 Profiling the Compiler

To evaluate the effectiveness of our profiler for profiling large applications, we undertook the (somewhat incestuous) task of profiling the core of the Glasgow Haskell compiler (see Section 5.1), with the aim of identifying its inefficiencies and improving them. This is a particularly large Haskell program consisting of over 200 modules and 30,000 lines of code. The initial profiling experiments examined the performance of Version 0.13 of the Glasgow Haskell compiler.

7.1.1 Initial profiles

We first determine the total cost of each pass of the compiler by placing `scc` annotations around each of the major passes. Figure 7.1 shows the aggregate profile of the compiler compiling one of its own modules (`TcExpr.lhs`). Each cost centre (except `MAIN`) corresponds to a particular pass of the compiler (see Figure 5.1). The initial cost centre, `MAIN`, is attributed with the costs of processing the I/O request dialogue, since this is outside

the scope of the source program.

The compilation requires 240 seconds of profiled execution time on our SS10-41. 45% of that time is spent in the type checker and 25% in the renamer with a further 7% spent in the built-in name environments (`builtinEnv`) by the renamer. As these passes take up a large proportion of the execution time they are the obvious places to focus on when optimising the compiler.

7.1.2 Input space leak

A heap profile of the same compilation is shown in Figure 7.2. The most significant feature of the heap profile is the large amount of space occupied by `rdModule` across much of the compilation. We would expect this input to be discarded as we constructed the abstract syntax tree. Clearly this does not happen.

Figure 7.2 reveals that the input is discarded at the onset of code generation. The critical event that allows the input to be discarded is the opening of the output file. Examining the code we find that the module name, which is required to name the output file, is bound in a lazy pattern match with the input module.

```
(mod_name, absyn_tree) = cvModule (rdModule input_pgm)
```

The identification of this space leak is slightly embarrassing as an identical space leak had already been identified by Runciman and Wakeling in the Chalmers **hbc/lml** compiler using their heap profiler (Runciman & Wakeling [1992]). This type of lazy pattern matching space leak can be eliminated if the evaluation mechanism (or garbage collector) arranges for *all* the pattern selectors to be evaluated when the first of them is evaluated (Sparud [1993]; Wadler [1987]).

Since `ghc` has no such evaluation mechanism we recoded the problematic lazy pattern match with a strict `case` expression. The resulting profile is shown in Figure 7.3. The input (now comprised of `rdImports` and `rdModule`) is consumed by the subsequent pass of the compiler. The space-time product has been reduced from 580Mbs to 255Mbs — a reduction of over 50%. This measure should be treated with caution since the improvement here is solely due to a reduced heap occupancy — there is no reduction in execution time.

Of course, space usage does have an indirect time cost, because it increases garbage-collection overheads (which are not included in the profiled execution timings). With

```
        Sat Jun 26 11:52 1993 Time and Allocation Profiling Report
                     (Lexical Scoping)

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

        total time  =       240.48 secs   (12024 ticks @ 20 ms)
        total alloc = 619,779,088 bytes  (51846277 closures)

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

Figure 7.1: Aggregate Profile (Version 0: `TcExpr.lhs`)

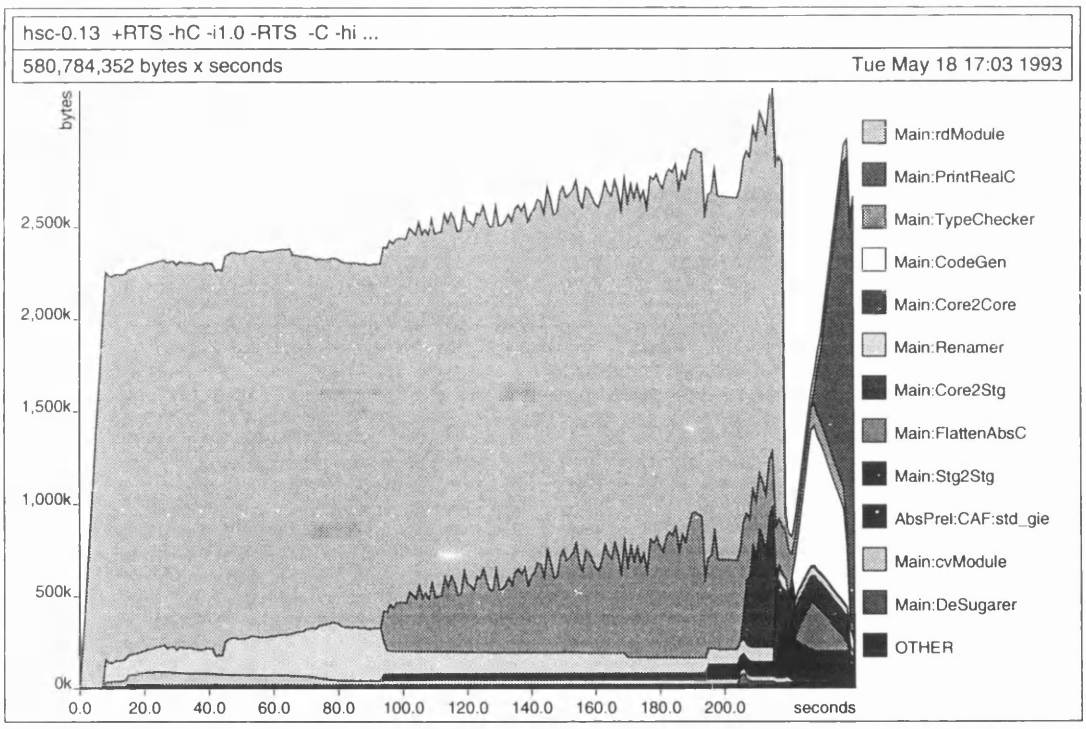


Figure 7.2: Heap Profile (Version 0: `TcExpr.lhs`)

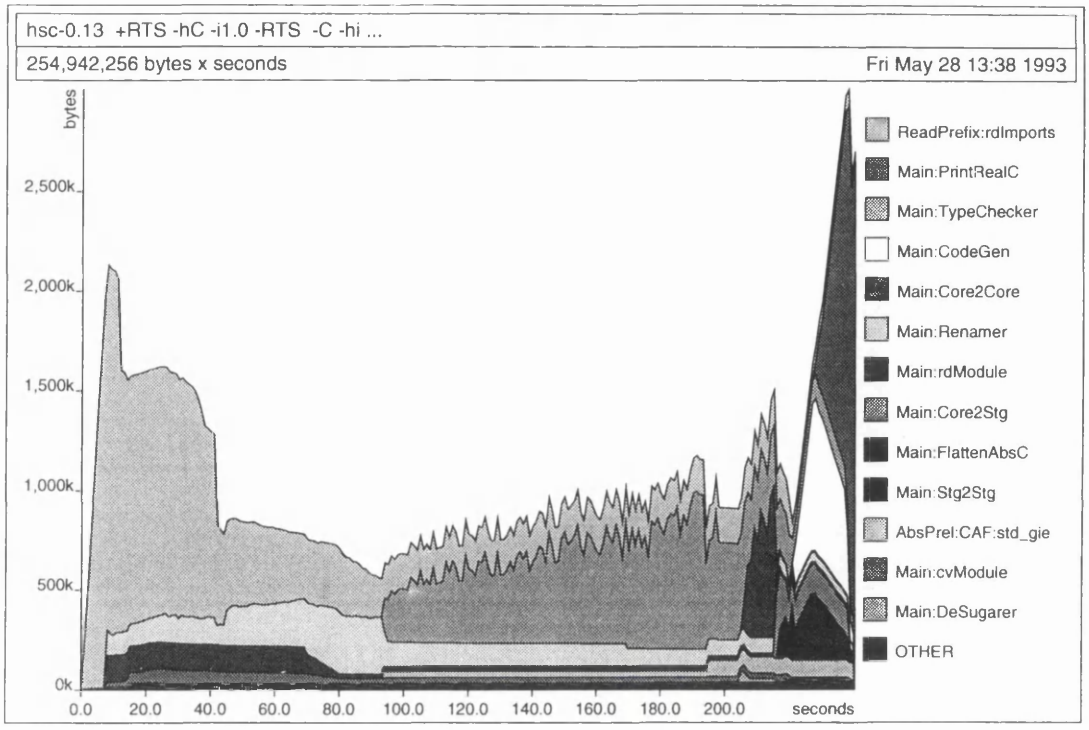


Figure 7.3: Heap Profile (Version 1: `TcExpr.lhs`)

traditional collection schemes, such as copying, this indirect cost is proportional to the space occupied. However, generational garbage collection reduces the impact of large space occupancy by promoting any long lived data and only collecting it occasionally. Unprofiled compilation of `TcExpr.lhs` with a 10Mb heap imposes a generational GC overhead of about 10%[1]. Reducing the space requirements can only improve the compilation time by a fraction of this amount.

During our investigation of the input space leak we added a `rdImports` cost centre which distinguishes the source of the interface files from the source of the module itself. The interface files are inserted into the source as each `import` statement is processed by the Haskell parser (see Figure 5.1). Since this is before the module source all the interface files must be read before the module source is encountered and compilation can proceed. This accounts for the large `rdImports` spike in the initial stages of compilation (`TcExpr.lhs` imports some large interface files). It may be possible to avoid this spike by placing the

[1]This 10% garbage-collection overhead assumes the machine has enough physical memory to avoid paging. If paging overheads are significant, reducing the space requirements to enable efficient execution with a smaller heap can result in substantial reductions in elapsed execution time.

```
      Sun Jun 27 21:46 1993 Time and Allocation Profiling Report
                  (Lexical Scoping)

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

      total time  =       230.20 secs   (11510 ticks @ 20 ms)
      total alloc = 619,779,216 bytes  (51846289 closures)

COST CENTRE      MODULE      GROUP          scc subcc  %time %alloc
extendSubst      Subst       basicTypes    1304 28768   35.5   43.6
buildLookupFn    NameEnv     envs          2617     0   28.3   38.0
PrintRealC       Main        main             1     0    4.9    4.3
lookupSubst      Subst       basicTypes   30678     0    4.8    0.0
Rename           Main        main             1  2617    3.9    3.4
CoreSimplify     SimplCore   simplCore        2  9542    3.9    2.0
TypeChecker      Main        main             1  4035    3.5    0.8
  ...
```

Figure 7.4: Further Time Profile Breakdown (Version 1: `TcExpr.lhs`)

interface files after the module source and/or forcing the processing of each interface file as it is read, but we have not attempted this.

7.1.3 Execution hot spots

Figure 7.1 revealed two execution hot spots: the type checker and the renamer. However further investigation is still required to identify the cause of the inefficiencies.

For the type checker we suspected that the inefficiencies were due to inefficient substitution algorithms based on a simple association list, but had never previously been able to quantify this. Annotating each of the functions in the substitution module reveals that nearly 36% of the entire compilation time is spent extending the substitution (a routine consisting of only 30 lines of code), with an additional 5% of the execution time spent searching the association list for a type variable's substitution (see Figure 7.4).

Once the extent of the substitution inefficiencies were quantified we decided that it would be worth investing the time to develop improved substitution algorithms that used a mutable array data structure. This is described in Section 7.1.6. First, though, we address the inefficiencies in the renamer.

7.1.4 The renamer

The job of the renamer is to resolve the scoping of source identifiers, replacing them with unique integers. We suspected that string-based lookups in name environments were consuming a lot of the time. Annotating the function which constructed and returned the function to look up names in an environment reveals that a total of 28% of the entire compilation time is spent building environments and looking up strings within them (see Figure 7.4). More significantly, the function `buildLookupFn` is called a total of 2617 times, resulting in the construction of 2617 environment lookup functions! This is very suspicious, since there are only 47 environments required (7 explicit environments plus 2 for each of 20 modules imported). This certainly does not account for the construction of 2617 environment lookup functions!

The current implementation of the lookup environments uses different algorithms depending on the number of elements in the environment.

- A simple unordered list search is used for small (less than 8 element) environments.
- If the environment is already sorted a binary tree is constructed.
- If not sorted a hash table (with 17 buckets) is constructed. This hash table is implemented as a list, indexed by the hash value (see Section 7.1.5).

Using cost centres to provide a breakdown of the costs associated with the different implementations produced the (partial) profile:

COST CENTRE	MODULE	GROUP	scc	subcc	%time	%alloc
mkHash	NameEnv	envs	2977	0	13.9	26.9
mkTree	NameEnv	envs	3983	0	13.1	11.0
lookupHash	NameEnv	envs	2977	0	0.3	0.1
buildLookupFn	NameEnv	envs	2617	14036	0.3	0.0
lookupTree	NameEnv	envs	3983	0	0.2	0.0
lookupList	NameEnv	envs	58	0	0.0	0.0
mkList	NameEnv	envs	58	0	0.0	0.0

These reveal that the environment lookup functions (`lookupHash`, `lookupTree` and `lookupList`) are called the same number of times as the corresponding functions which build the lookup data structures (`mkHash`, `mkTree`, and `mkList`)! One would have expected many lookups to be performed on each structure. Examining the STG-machine code (dumped by the compiler) we see that

```
mkGenericLookupFun_hash_tbl eq_k lt_k hash_k stuff
  = lookup_Hash eq_k lt_k hash_k (mk_Hash eq_k lt_k hash_k stuff)
```

is translated to

```
mkGenericLookupFun_hash_tbl eq_k lt_k hash_k stuff sat.T1
  = let hash_tbl = mk_Hash.wrk hash_k stuff
    in lookup_Hash.wrk eq_k hash_k hash_tbl sat.T1
```

This reveals an error in the argument saturation pass of the compiler: the `hash_tbl` is built only after, and every time, the lookup argument is supplied! After fixing this optimisation bug we found that the compiler still built 295 environments. Continued investigation revealed a second optimisation bug that duplicated work by substituting bindings inside anonymous lambda expressions.

Profiling the execution after fixing both these compiler bugs revealed a 34% (82 seconds) reduction in total execution time with the total renaming costs dropped from 78 seconds to just 4 seconds.

7.1.5 Hash tables

The work with the renamer drew our attention to a rather inefficient implementation of the name environment hash table — a list indexed by the hash value has an access time proportional to the hash value. We decided to develop a hash-table implementation based on *array transformers* (Peyton Jones & Wadler [1993]) and compare the efficiency with the indexed-list implementation.

The relative performance results are reported in Figure 7.5. These were gathered by profiling a number of compilations, each using a different hash table implementation, and comparing the time attributed to the `mkHash` and `lookupHash` cost centres. With 17 buckets the array implementation results in a smaller lookup time, but construction time actually increased slightly. This is because the cost of a read-write sequence with an array transformer is slightly higher than the average insertion cost into a 17 element index-list insert (average insert position 8.5).

The benefit of using arrays is the ability to use a larger number of buckets, reducing the lookup costs as there are fewer elements in each bucket, without incuring increased construction overheads.[2] With a hash-indexed list implementation, increasing the number

No. of Hash Buckets	Construction Cost		Lookup Costs	
	Indexed List	Array	Indexed List	Array
17	1.00	1.04	1.00	0.60
37	1.77	1.05	0.95	0.42
79	3.60	1.04	1.52	0.39

Figure 7.5: Hash Table Performance Comparison (relative to 17 bucket indexed list)

of buckets increases construction costs. Lookup costs also increase when the increased linear access costs outweigh the reduced bucket search costs (observe 79 buckets).

The array implementation (with 79 buckets) improved the performance of the hashed name environment by over 50%. (The lookup costs dominate, once repeated construction has been avoided.) However, since the name environments were no longer a bottleneck the overall impact of this improvement was very small — less than half a second.

7.1.6 The substitution

The implementation of the substitution was inefficient for two reasons:

- The lookup structure was based on a simple association list that had to be searched every time a type variable's substitution was required.
- The type being substituted for each type variable was stored idempotently — the type to be substituted for a type variable is applied to all the existing types in the substitution whenever the substitution is extended.

We decided to use monadic mutable array technology (Launchbury [1993b]; Wadler [1990]) to implement a graph-rewriting version of the substitution algorithm. This idea was proposed by Hammond in (Hammond [1991]), in response to an intuition that the substitution was a bottleneck within the compiler. (Our profiling results have confirmed and quantified Hammond's intuition.) At that time, an implementation was not practical since efficient array implementations were not available. Since then, support for mutable arrays has been added to the Glasgow Haskell compiler.

[2] Actually there is a small linear cost associated with increasing the array size as all the elements of the array must be initialised when the array is allocated.

Algorithm	Substitution	Total
Idempotent association list	98s	150s
Non-idempotent association list	18s	68s
Non-idempotent mutable array	1.8s	50s

Figure 7.6: Performance of monadised substitution algorithms (`TcExpr.lhs`)

The type checker already had a customised monad threaded through it. It was responsible for:

- Carrying the current substitution, a unique name supply, and the current source location.
- Catching, reporting and recovering from any type checking errors.

This monad had to be extended to enable the implementation to be modified to use a mutable array. The following modifications were required:

- The monad was threaded through the unifier (previously the substitution was passed explicitly through the unifier).
- The monad interface was extended to provide the required substitution operations to the unifier.
- A special unique name supply used only by the type variables was added. This was used to directly index the substitution array. The array is dynamically resized if it overflows.

These modifications took a significant amount of time to implement. However once they were in it was a simple matter to change the implementation of the monad and experiment with different substitution algorithms.

We compared three different substitution implementations:

- An idempotent association list (the original implementation).
- A non-idempotent association list. This algorithm has to apply the substitution to the type being substituted before returning it.
- A non-idempotent representation stored in a mutable array. This provides constant-time lookup and modification.

The results in Figure 7.6 show quite spectacular speedups. Making the substitution representation non-idempotent improved the performance of the substitution algorithm by a factor of 5. This could have been undertaken without all the mutable-array modifications described above. However the modifications proved worthwhile as the introduction of a mutable array as the underlying data structure provided a further 10 times speedup. Overall the performance of the substitution algorithm was improved by a factor of more than 50!

7.1.7 Overall improvement

The time profile of the optimised compiler (Version 2) is shown in Figure 7.7. Comparison to Figure 7.1 shows an overall reduction in execution time of 79%, with total execution time dropping from 240 seconds to 50 seconds. Figure 7.7 reveals a much more balanced time profile with no unexpected inefficiencies — though `PrintRealC` and `CoreSimplify` still look like good candidates for optimisation. The dominant compilation tasks are now I/O related as the summary in Figure 7.8 reveals.

The heap profile for Version 2 of the compiler is shown in Figure 7.9. This is dominated by a peak at the end of compilation which we suspect is a space leak in the code generator. Comparison with the initial heap profile (Figure 7.2) reveals a slight increase in the peak memory requirements, but an overall reduction in the space-time product of 85%, from 580Mbs to 81Mbs.

The module we were profiling had a particularly "hard" type checking problem. This resulted in spectacular overall performance improvements when the substitution algorithm was improved. Figure 7.10 gives a summary of the performance improvements for the compilation of all 211 modules that make up the compiler. The reduction in execution time for the compilation of all modules was 51%.

The compilation of some modules is dominated by inefficiencies that were not revealed by the profiling of `TcExpr.lhs`. For example, the compilation of one 1200 line module, `AbsPrel.lhs`, accounts for over 10% of the total time to compile the 30,000 line compiler! This is due to inefficiencies in the optimisation and analysis phases of the compiler when presented with a very large static data object. Further investigation is required — at least

```
      Tue Aug 17 14:13 1993 Time and Allocation Profiling Report
                    (Lexical Scoping)

        hsc-0.13 +RTS -p -hC -i0.5 -RTS  -C -hi ...

      total time  =        50.46 secs   (2523 ticks @ 20 ms)
      total alloc =  97,950,284 bytes   (8076271 closures)

COST CENTRE      MODULE      GROUP        scc subcc  %time %alloc
PrintRealC       Main        main           1     0   21.4   26.5
CoreSimplify     SimplCore   simplCore      2     0   13.9   13.0
MAIN             MAIN        MAIN           1     1   11.4   17.4
TypeChecker      Main        main           1     0   10.2   11.3
Renamer          Main        main           1     0    7.0    3.9
rdImports        ReadPrefix  reader         1     0    6.5    6.9
CodeGen          Main        main           1     0    5.6    6.3
FlattenAbsC      Main        main           1     0    3.0    3.2
cvModule         Main        main           1     1    2.7    3.5
Core2Stg         Main        main           1     0    1.7    1.3
StgFloat         SimplStg    simplStg       1     0    1.3    1.1
CoreStranal      SimplCore   simplCore      1     0    0.9    0.7
DeSugarer        Main        main           1     0    0.8    0.7
StgUpdAnal       SimplStg    simplStg       1     0    0.7    0.6
builtinEnv       Main        main           1     0    0.6    0.1
rdModule         Main        main           1     1    0.4    0.3
  ...
```

Figure 7.7: Time Profile (Version 2: `TcExpr.lhs`)

Task	Time	Components
Input/Output	40%	`PrintRealC rdImports rdModule MAIN` (actual character I/O)
Optimisation and Analysis	17%	`CoreSimplify CoreStranal StgFloat StgUpdAnal`
Type Checking	10%	`TypeChecker` (including the substitution)
Code Generation	9%	`CodeGen FlattenAbsC`
Renaming	8%	`Renamer builtinEnv`
Translation	5%	`cvModule DeSugarer Core2Stg`
Other	1%	
CAFs	10%	(lexical scoping CAF/dictionary costs)

Figure 7.8: Summary of Time Profile (Version 2: `TcExpr.lhs`)

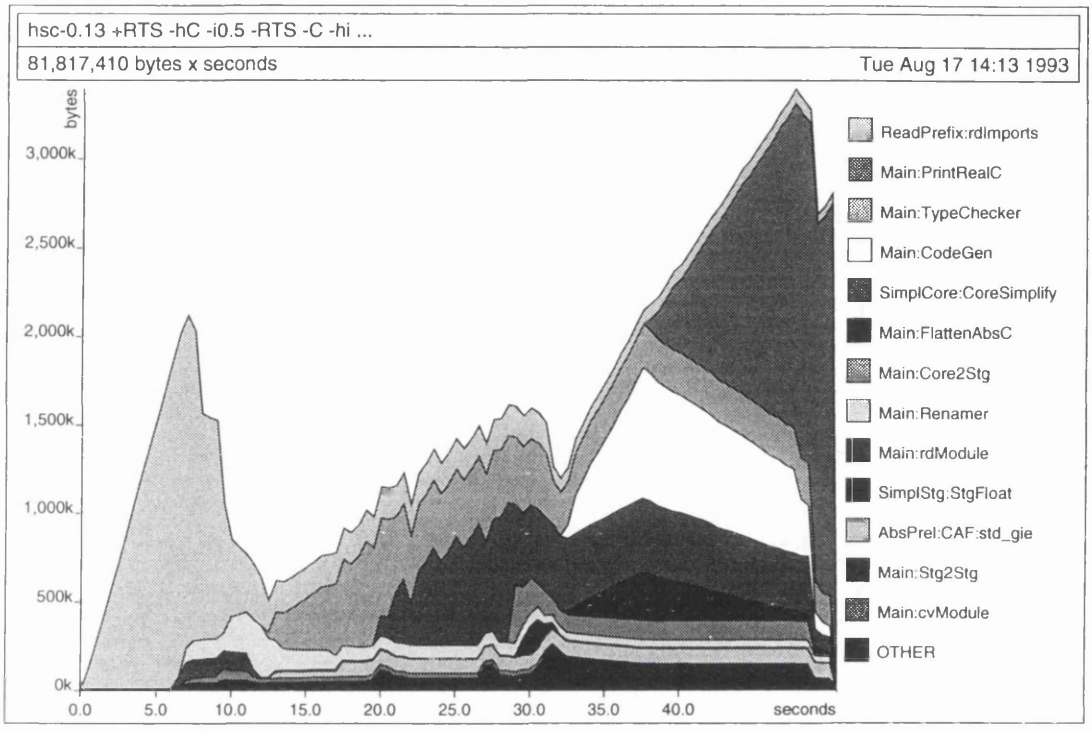


Figure 7.9: Heap Profile (Version 2: `TcExpr.lhs`)

Module	Initial	Improved	Reduction
`TcExpr.lhs` (best)	240s	50s	79%
`AbsPrel.lhs`	1499s	1015s	32%
`PrefixSyn.lhs` (worst)	19s	16s	16%
TOTAL (211 modules)	15612s	7604s	51%

Figure 7.10: Performance Improvements Compiling the Whole Compiler (`-O`)

we now have the tools to undertake this.[3]

7.2 Other Applications

A version of the lexical profiler has been distributed with the Glasgow Haskell compiler since Version 0.15 (June 1993) enabling other users to profile their Haskell applications. This has provided us with invaluable feedback about the practical use of the profiler for profiling large applications. This section reports some of the experiences other users have had using the **ghc** time profiler to profile their applications. I am very grateful to Julian Seward and Stephen Jarvis for their cooperation in providing considerable feedback about their experiences using the profiler.

7.2.1 Profiling a strictness analyser

Seward [1994] used an early version of the profiler to examine the internal dynamics of his 12,000 line, frontier-based strictness analyser. Explicit `scc` annotations were used to gather information about a large number of functional components in which he was interested. He then wrote a very simple post-processor (in Haskell) that summed the costs of all related components, as specified by an auxiliary input file. This enabled him to determine the costs of the different "parts" of the algorithm, without having to remove all his low-level `scc` annotations and recompiling. He used these results to compare the performance of a number of different analysis techniques.

Problems encountered

Unfortunately, Seward found that up to 40% of execution time was being attributed to CAF (and DATA) cost centres. This highlighted the problems with lexical scoping and prompted us to develop our hybrid profiling scheme (Section 4.4).

The fact that Seward needed to develop a customised inheritance post-processor indicates that extending our implementation with statistical inheritance (Section 8.4.2) may be a very worthwhile enhancement.

[3] Subsequent improvements to the strictness analyser have reduced the performance problems encountered when compiling `AbsPrel.lhs` and other programs that contain large static data objects.

7.2.2 Profiling a natural language processor

More recently, the profiler has been used to profile and improve the performance of LOLITA (Large-scale, Object-based, Linguistic Interactor, Translator and Analyser) (Garigliano, Morgan & Smith [1992]; Jarvis [1994]) — a natural language system which has been developed by the Artificial Intelligence Research Group at the University of Durham. This consists of approximately 30,000 lines of Haskell, divided into 150 modules.

Compilation using the Glasgow Haskell compiler has been in operation for a number of months, with the emphasis on using the time profiler to identify bottlenecks and inefficiencies in the code. As well as improving the performance of Lolita, the study also investigated different methods of profiling.

Initial results were obtained using the `-auto-all` option to annotate all the top-level declarations. This highlighted the basic functions in the system that were responsible for a large proportion of execution time. By identifying and improving these particular functions, which were generally simple operations called hundreds of thousands of times, they were to bring about an initial performance improvement of 7.8%. This task required no detailed understanding of the application being optimised.

Further performance improvements were obtained by developing more efficient algorithms for system components that consumed a large proportion of the execution time. These improvements required a more detailed understanding of the application being profiled.

The Lolita system uses a hash dictionary to perform efficient word lookup. The system computes a hash number from a given input word and this number refers to where the word is located in the Lolita dictionary. This feature is a key operation in the Lolita system so it was not surprising to find the hash operation high in the profiling statistics. Using the profiler to record progressive changes the hashing function was rewritten to a benefit of a further 11.7%, giving a compound improvement of 19.3%. Further work in this area is currently in progress, to move from a hash-table implementation to a tree. At each node in this tree either an array or a list is used to store the subtrees depending upon the number of subtrees below it. The profiler is being used to calculate the optimal point at which we convert from list representations to arrays, i.e. the threshold at which array access is quicker than list access, and thus providing the optimal look up time for a word.

Although the integration of this implementation with Lolita has yet to be completed, the profiling results indicate a fifteen fold improvement on the old method. When this work is completed impressive improvements to the word look up time of the system are expected.

Lolita is built around a large semantic network that holds information and data about the world as well as some of its linguistic data. The semantic network consists of over 35,000 conceptual graph nodes capable of representing over 100,000 inflected word forms. The representation and accessibility of the network is therefore an essential point in the efficiency of the system. Profiling information had identified the indexing and update operations of the semantic net as costly, accounting for 30-40% of all system costs. Moving away from the automatic annotation to explicit source annotation enabled all these costs to be attributed to just two cost centres, reporting the total indexing and update costs respectively (although problems with costs being attributed to CAF cost centres were encountered).

The semantic network was originally loaded into a collection of single dimensional list structures, from which data could be accessed and updated. This single dimensional list structure has evolved through a 2D array to an n-arry-tree of arrays, the latter containing a method by which the size of the leaf nodes, storing the data in arrays, can be offset against the depth of the tree. Profiling has played a key role in the development and evaluation of these algorithms. The final tree representation of the semantic network brought about improvements of 15-20%. Compound improvements now stand at 35.4% and many further improvements are envisaged.

Problems encountered

The major problem encountered while using the profiler was the attribution of costs to CAF cost centres. This often accounted for 30% of the profiling results. The overheads of the profiler were also a problem, both in terms of the increased time needed for program execution with the profiling options set, and also the amount of heap space needed.

The space problems were not surprising since normal execution required 42Mb resident heap. The corresponding profiled execution has a 75Mb heap residency which requires a heap size of about 160Mb when using the two-space copying collector. There is a clear need to reduce the excessive space requirements. We are currently planning to develop a

version of the profiler which uses our inplace generational garbage collector (Sansom & Peyton Jones [1993]). Building a version of the profiler that does not store the creation time in each closure would also reduce the space overhead. The time problems are still being investigated.

7.3 Conclusion

The **ghc** profiler has been successfully used to profile and improve the performance of a number of large applications. Most notably, the compiler itself, where a 51% performance improvement was achieved, and Lolita, a natural language system, where a 35% performance improvement has been achieved to date.

7.3.1 Using the Profiler

Most users have tended to use the time profiler to identify and improve execution hotspots. We believe this is because the time profile provides a form of feedback that is more tangible since it can be directly related to the observed execution time. The time profiler has proved particularly useful for the following tasks:

- Identifying the basic functions that are responsible for a large proportion of execution time (using automatic annotation). These are generally simple operations that are called hundreds of thousands of times.
- Identifying the system components that consume the majority of the execution time (using explicit source annotations).
- Quantifying the potential benefits of an improvement, before a complete implementation is undertaken. This provides a sound basis for deciding if a proposed improvement might be worthwhile.
- Evaluation and comparison of different algorithmic solutions for a particular system component.
- Tuning the performance of a particular algorithm.

The major problem encountered while using the lexical profiler was the attribution of function costs to CAF cost centres. Implementing the proposed hybrid profiling scheme

(Section 4.4.3) should significantly improve the usability of the profiler.

The heap profiler has not been used as extensively as the time profiler. We have found that, once provided with a time profiler, users do not use the heap profiler, unless the space requirements are causing significant performance degradation due to thrashing or physical memory constraints.

7.3.2 Diagnosing performance bugs

It is not enough to simply profile and identify the execution hot-spots. The cause of the inefficiencies must also be identified before they can be addressed. This may require more specific information about the dynamic behaviour of the algorithm being executed. The profiler does provide some dynamic information, in the form of `scc` counts and the serial profiles, but additional diagnostic or debugging tools are also needed.

One very simple diagnostic tool that the ghc compiler provides is a side-effecting `trace` "function". When entered, `trace` evaluates and prints its first argument on `stderr` and returns the value of its second argument. It is a primitive, but useful, debugging tool since it can be used to reveal specific information about the dynamic execution, in much the same way as informational `print` statements are often used in conventional languages. Unfortunately `trace` affects the evaluation order — forcing the evaluation of its first argument. More work still needs to be done developing more sophisticated diagnostic and debugging tools.

A particularly awkward diagnostic problem is identifying the cause of a space leak. The heap profiles address the question "What is in the heap?", but they do necessarily help with the question "Why is it in the heap?". Further work needs to be done to develop tools that help to identify "What is holding onto the closures in the heap?". The development of the **nhc** *retainer* profiler is an encouraging step.

Chapter 8

Conclusions

This research set out to develop a practical time and space profiler for a lazy higher-order functional language which relates the profiling data back to the original source in a way that is meaningful to the programmer. On the way we encountered some rather subtle issues concerning the attribution of execution costs. This led to the development of a formal semantics of cost attribution that has proved invaluable in providing insight and enabling a precise formulation of the distinction between two different cost attribution schemes: lexical scoping and evaluation scoping. Given this framework, the subsequent development of the hybrid profiling semantics proved almost trivial.

Associating a cost centre with each profiled expression, using the `scc` construct, has proved very convenient. As well as preventing "bad" transformations, it provides a language in which cost preserving transformations can be expressed. Parts of the original expression can be moved into the scope of a different cost centre provided they are annotated with their original cost centre. When no `scc` annotations are present program optimisation proceeds as normal.

The formal approach was also used to specify equivalent abstract cost semantics based on the push-enter graph-reduction model of evaluation. The conversion to the push-enter semantics highlighted a set of implementation related design decisions made on the way to our STG-machine implementation. However these semantics and the design decisions that they highlighted are applicable to a number of different abstract machines based on graph reduction, such as the G-machine and the TIM.

The final step in our implementation, mapping these semantics onto the STG-machine,

then proved quite straightforward. This was again specified in a formal manner by extending the state transition semantics with the required manipulation cost centres.

The incorporation of the profiler into the Glasgow Haskell compiler has demonstrated the practicality of our approach since it has enabled large application programs to be profiled and improved. It has also provided invaluable feedback from real users. Most notably, it revealed that the lexical scoping attribution of function costs to CAF cost centres was a serious practical problem. This prompted the development of the hybrid profiling scheme which should significantly improve the usability of the profiler.

8.1 Current Status

A version of the lexical profiler has been distributed with the Glasgow Haskell compiler since Version 0.15 (June 1993). However, this implementation has a couple of shortcomings:

- The boxing transformation described in Section 4.2.4 is not implemented. The cost of top-level functions that are passed as arguments are attributed to the application site, not the reference site.
- Many of the `scc` specific transformations are not implemented. Most notably, $\texttt{scc}_{sub}$ annotations within the transformation passes of the compiler have not been implemented. Consequently the current implementation does not unfold declarations inside an `scc` annotation.

We intend to complete the implementation work for the next public release of the compiler (Version 0.22). In particular we plan to:

- Release a version of the hybrid profiling scheme.
- Implement the boxing transformation required by the lexical and hybrid profilers.
- Introduce $\texttt{scc}_{sub}$ annotations to enable more program transformation in the presence of `scc` annotations. The most important of these is the unfolding of declarations inside `scc` annotations and the floating of `let`-bindings in and out of `scc` annotations.
- Introduce $\texttt{scc}_{dict}$ annotations and the automatic annotation of dictionary construction.

- Improve the naming of local bindings introduced by the compiler.
- Report the garbage collection time and estimated maximum residency in the cost centre profile report (Section 6.2). This should draw the attention of the user to any unreasonable space costs.

This work should greatly improve the usability of the profiler.

8.2 Continuing Development

Aside from completing the implementation for the next release, there are a number of developments to the profiler that we are currently considering.

- The current implementation of the profiler uses the two-space garbage collector. This imposes a 100% space overhead for the second semi-space. Developing a profiling runtime system that is based on the generational garbage collector will remove this overhead. Data for the heap profiles can be gathered using the mark phase of the major collection.
- Providing a mechanism to enable cost centres to be activated and deactivated at runtime. This should significantly reduce the amount of recompilation required during profiling.
- Implementing specific transformations that deal with the situation where a particular transformation is hindered by an intervening `scc` annotation.
- Introducing the `get_ccc` primitive to enable the enclosing cost centres to be determined dynamically (Section 5.4.4).
- Developing a serial profile reporting the distribution of closure entry counts. This would be more precise than the serial time profile we currently produce (Section 6.4).
- The current heap profiles identify "what is in the heap", not "why it is in the heap". Developing heap profiles, like the **nhc** *retainer* profile, that identify "what is holding onto the closures in the heap" should aid the difficult task of tracking down the cause of a space leak.

In addition we always attempt to respond to any feedback received from our users.

Unfortunately, the incorporation of the profiler in the Glasgow Haskell imposes a cost on subsequent development of the compiler since all developments must now ensure that the attribution of costs is preserved when compiling for profiled execution.

8.3 Formalism in Practice

I believe that the fairly formal approach to the attribution and measurement of profiled costs is a distinctive contribution of this thesis. It is interesting that this formalism emerged within the context of a practical project as a tool for managing the intellectual complexity of a real problem. The formalism itself was not the goal of this research.

As in almost all formalism it provides an abstraction from some, but not all, implementation issues. Different layers of formalism, each providing a different level of abstraction, were used to isolate different design issues. The design decisions made at one level of abstraction provided the basis for the more detailed formalisms subsequently developed.

In hindsight one might argue that introducing a formal approach earlier in the development of the profiler could have identified some of the more significant issues earlier. However, I am not convinced that this would be the case since experience from our initial implementation provided a lot of input into the development of the abstract cost semantics. The result was a formalism with the appropriate level of abstraction, identifying exactly what we needed, without incorporating unnecessary detail. The necessary detail was then incrementally exposed once the major design issues had been identified.

8.4 Future Directions

There are a number of possible directions for future work, both theoretical and practical, which are discussed in the following sections.

8.4.1 Formal proofs

The formal semantics of cost attribution developed here could provide the basis for proving certain properties about the correctness of the implementation. For example, using the abstract cost semantics as a definition of the required attribution of costs one would like

to prove that the compiler transformations are indeed faithful to that cost attribution.

The equivalence of the abstract cost-semantics, the push-enter semantics and the STG state transition system could also be investigated. We are currently working on proving the equivalence of the abstract cost semantics and a cost-augmented abstract state transition system.

8.4.2 Inheritance profiling

A more practical future development would be to extend the profiler to incorporate a form of statistical inheritance.

The profiler currently produces a flat profile, with costs only being attributed to the immediately enclosing cost centre. However, it is possibly to gather profiling data that would enable the statistical inheritance of costs up the reference graph.

The required profiling data can be gathered by attributing profiling data to a pair of cost centres.[1] Each cost-centre pair contains:

- The current cost centre, and
- The cost centre that enclosed the `scc` annotation which set the current cost centre i.e. the cost centre one arc up the reference graph.

The current cost centre register would become a current cost-centre-pair register. This cost-centre pair is stored in each closure when it is allocated. Profiling data, such as `scc` counts, time ticks, and allocation, can then be attributed to the current cost-centre pair, and the profiling output extended to report the more detailed cost-centre pair information. A statistical inheritance post-processor could then be developed.

The main problem with this scheme is the construction of cost centre pairs. When an `scc` expression is evaluated a cost-centre pair, containing the enclosing cost centre and the `scc` cost centre, must be loaded into the current cost-centre-pair register. Since the enclosing cost centre may not be known at compile time the cost-centre pair can only be determined at runtime. Unfortunately, this necessarily introduces some dynamic execution, whenever an `scc` is evaluated, increasing the profiling overhead.

[1] The notion of a cost-centre pair is remarkably similar to the colour pairs used by the UCL profiler (Section 3.3.3)

Fraser & Hanson [1991] describe a solution to exactly this problem in the context of their C compiler that is easily adapted to our cost centre implementation. Each cost centre is linked to all the cost-centre pairs in which it is the current cost centre. When an `scc` is evaluated this list is searched for the pair containing the enclosing cost centre, extracted from the previous cost-centre pair. If it is not found a new cost-centre pair structure is allocated from a pre-declared array, and added to the list for that cost centre. Various optimisations are possible, such as a special test for cost-centre pairs that have the same current and enclosing cost centres. These arise in recursive functions that contain an `scc` annotation.

The modifications required to implement cost-centre pairs are limited to the C macros that manipulate cost centres and record profiling data, and the runtime system. No modifications need to be made to the compiler. It generates exactly the same code to manipulate cost-centre pairs as it currently does to manipulate ordinary cost centres.

8.4.3 Programming environment

Generally programmers only resort to profiling when they encounter a noticeable performance problem. Routinely generating profiles to examine the behaviour of the program you have just written is the exception rather than the rule. As most programmers are surprised by the contents of the profile when they do bother to profile their program, providing an integrated programming environment that automatically generated profiles, using automatic `scc` annotation, and presented them to the programmer, could improve the understanding programmers have about the programs they write. It could also result in increased productivity as programmers could concentrate on writing correct programs, knowing that they will be presented with a profile that will direct them to any execution bottlenecks.

Of course, more detailed, programmer directed profiling, could then be undertaken once the existence of a performance problem was drawn to the attention of the programmer.

Profiling in real time

As part of the automatic generation of profiles the runtime system could be extended to display the profiling data to the user while the program is executing. The serial profiles

are ideally suited for this. They could be drawn in a separate window as the program executes, providing immediate feedback to the programmer. This is particularly useful when profiling interactive programs since the programmer can observe the effect of a particular interaction on the execution the program as it occurs. A thorough treatment of this topic can be found in Jeffrey [1993].

8.4.4 Parallel profiling

This thesis has not addressed the profiling of the parallel execution of lazy functional programs. However, the cost centre model could be used to profile parallel execution. Each processor could record information about the execution it performs, attributing the costs to a cost centre local to the processor. Separate profiles of the execution activity of each processor could then be presented, or the data could be combined into a global execution profile. Parallel execution overheads, such as time spent communication between processors and idle time, could be attributed to special cost centres and reported as part of the profile. This approach is being explored by Clack, Clayman & Parrott [1994] who intend to extend the UCL profiler to profile parallel execution on the DIGRESS system at Athena Systems Design Ltd.

8.5 Final Remark

A major attraction of this research has been the very tangible benefit to the practical development of lazy functional programming. Three years ago there were virtually no profiling tools available for lazy functional languages. Understanding what was going on inside them was more of an art than a science. We now have profiling tools that are comparable to, and in many cases, a lot better than, the profiling tools available for conventional languages. This is a very encouraging situation. We hope that it will aid the use of lazy functional languages for real applications programming.

Bibliography

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

J Armstrong [1993], "Industrial Experience of Declarative Programming," Computer Science Laboratory, Ellemtel Communications Systems Laboratories, Älvsjö, Sweden.

L Augustsson [June 1993], "Implementing Haskell overloading," in *Functional Programming Languages and Computer Architecture, Copenhagen*, ACM.

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

DR Barach & DH Taenzer [May 1982], "A technique for finding storage allocation errors in C-language programs," *SIGPLAN Notices* 17, 16–21.

JF Bartlett [Feb 1988], "Compacting garbage collection with ambiguous roots," Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, Technical Report 88/2.

JL Bentley [1982], *Writing Efficient Programs*, Prentice Hall.

JL Bentley [July 1987], "Programming Pearls — Profilers," *Communications of the ACM* 30, 587–592.

RD Bergeron & HR Bulterman [March 1975], "A technique for evaluation of user systems on an IBM S/370," *Software — Practice and Experience* 5, 83–92.

H Boehm & M Wuiser [Sept 1988], "Garbage collection in an uncooperative environment," *Software – Practice and Experience* 18, 807–820.

DF Brailsford, E Foxley, KC Mander & DJ Morgan [June 1977], "Runtime profiling of Algol 68-R programs using DIDYMUS and SCAMP," *SIGPLAN Notices* 12, 27–33.

P Caplinger [Feb 1988], "A memory allocator with garbage collection for C," in *Proceedings of the Winter 1988 USENIX Conference, Dallas, Texas*, 325–330.

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

CA Coutant, RE Griswold & DR Hanson [Jan 1983], "Measuring the performance and behaviour of Icon programs," *IEEE Transactions on Software Engineering* SE-9, 93–103.

SC Darden & SB Heller [Oct 1970], "Streamlining your software development," *Computer Decisions* 2, 29–33.

Jon Fairbairn & Stuart Wray [Sept 1987], "TIM - a simple lazy abstract machine to execute supercombinators," in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, G Kahn, ed., Springer-Verlag, LNCS 274, 34–45.

E Foxley & DJ Morgan [Jan 1978], "Monitoring the runtime activity of Algol 68-R programs," *Software — Practice and Experience* 8, 29–34.

CW Fraser & DR Hanson [Oct 1991], "A retargetable compiler for ANSI C," *ACM SIGPLAN Notices* 26.

R Garigliano, RG Morgan & MH Smith [Sept 1992], "LOLITA: Progress Report 1," Technical report 12/92, Artificial Intellegence Research Group, University of Durham.

A Gill, J Launchbury & SL Peyton Jones [June 1993], "A short cut to deforestation," in *Functional Programming Languages and Computer Architecture, Copenhagen*, ACM.

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

PW Grant, JA Sharp, MF Webster & X Zhang [June 1993], "Some issues in a functional implementation of a finite element algorithm," in *Functional Programming Languages and Computer Architecture, Copenhagen*, ACM.

C Hall, K Hammond, SL Peyton Jones & P Wadler [Jan 1994], "Type Classes in Haskell," Research Report FP-94-04, Dept of Computer Science, University of Glasgow.

K Hammond [Aug 1991], "Efficient type inference using monads," in *Functional Programming, Glasgow 1991*, R Heldal, CK Holst & P Wadler, eds., Springer-Verlag, Workshops in Computing, Portree, Scotland.

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

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

RJM Hughes [July 1983], "The design and implementation of programming languages," PhD thesis, Programming Research Group, Oxford.

D Ingalls [1972], "The execution profile as a measurement tool," in *Design and Optimisation of Compilers*, R Ruskin, ed., Prentice Hall, 107–128.

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

S Jasik [1972], "Monitoring program execution on the CDC 6000 series machines," in *Design and Optimisation of Compilers*, R Ruskin, ed., Prentice Hall, 129–136.

CL Jeffrey [1993], "A framework for monitoring program execution," PhD Thesis, TR 93-21, Dept of Computer Science, University of Arizona.

MP Jones [1992], "Efficient implementation of type class overloading," Dept of Computer Science, Oxford University.

AS Kishon [1992], "Theory and art of semantics-directed program execution monitoring," PhD Thesis, Dept of Computer Science, Yale University.

DE Knuth [1971], "An Empirical Study of FORTRAN Programs," *Software — Practice and Experience* 1, 105–133.

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

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

J Launchbury [June 1993b], "Lazy imperative programming," in *Proceedings of ACM Sigplan Workshop on State in Programming Languages, Copenhagen*, (available as YALEU/DCS/RR-968, Yale University), 46–56.

J Launchbury, A Gill, J Hughes, S Marlow, SL Peyton Jones & P Wadler [July 1992], "Avoiding Unnecessary Updates," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland.

G Lyon & RB Stillman [Oct 1975], "Simple transformations for instrumenting FORTRAN decks," *Software — Practice and Experience* 5, 347–358.

S Marlow [July 1993], "Update avoidance analysis by abstract interpretation," in *Functional Programming, Glasgow 1993, draft proceedings*, Dept of Computer Science, University of Glasgow, Ayr, Scotland.

S Matwin & M Missala [Aug 1976], "A simple machine independent tool for obtaining rough measures of Pascal programs," *SIGPLAN Notices* 11, 42–45.

SL Meira [March 1985], "On the efficiency of applicative algorithms," PhD thesis, Univ of Kent, Canterbury.

RL Page & BD Moe [June 1993], "Experience with a large scientific application in a functional language," in *Functional Programming Languages and Computer Architecture, Copenhagen*, ACM.

D Parrott & S Clayman [Nov 1990], "Report on 'Cost' and 'Debug' primitive extensions to FLIC," Technical Report, Dept of Computer Science, University College London.

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

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

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

SL Peyton Jones & J Launchbury [Sept 1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture, Boston*, LNCS 523, Springer Verlag.

SL Peyton Jones & D Lester [1990], "A Modular fully-lazy lambda lifter in Haskell," CSC 90/R17, Dept of Computer Science, University of Glasgow.

SL Peyton Jones, A Santos & W Partain [1994], "Let-floating: a modest transformation with big effects," Dept of Computer Science, University of Glasgow.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *Proc 20th ACM Symposium on Principles of Programming Languages, Charlotte*, ACM.

GD Ripley [Jul 1977], "Program Perspectives: A relational representation of measurement data," *IEEE Transactions on Software Engineering* SE-3, 296–300.

GD Ripley & RE Griswold [May 1975], "Tools for measurement of SNOBOL4 programs," *SIGPLAN Notices* 10, 36–52.

GD Ripley, RE Griswold & DR Hanson [March 1978], "Performance of storage management in an implementation of SNOBOL4," *IEEE Transactions on Software Engineering* SE-4, 130–137.

N Röjemo [Jan 1994], "nhc — Nearly a Haskell compiler," in *Proceedings of La Wintermote*, Dept of Computer Science, Chlamers University, Sweden.

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

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

C Runciman & D Wakeling [Aug 1990], "Problems and proposals for time and space profiling of functional programs," in *Functional Programming, Glasgow 1990*, SL Peyton Jones, G Hutton & CK Holst, eds., Springer-Verlag, Workshops in Computing, Ullapool, Scotland.

C Runciman & D Wakeling [July 1992], "Heap profiling of a lazy functional compiler," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland.

P Sanders & C Runciman [July 1992], "LZW text compression in Haskell," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland.

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

PM Sansom & SL Peyton Jones [July 1992], "Profiling lazy functional programs," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland.

PM Sansom & SL Peyton Jones [June 1993], "Generational garbage collection for Haskell," in *Functional Programming Languages and Computer Architecture, Copenhagen*, ACM.

A Santos & SL Peyton Jones [July 1992], "On program transformation in the Glasgow Haskell compiler," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland.

E Satterthwaite [1972], "Debugging tools for high level languages," *Software — Practice and Experience* 2, 197–217.

J Seward [1994], "Abstract Interpretation: A Quantitiative Assessment," PhD thesis in preparation, Dept of Computer Science, University of Manchester.

RL Sites [Dec 1978], "Programming Tools: Statement counts and procedure timings," *SIGPLAN Notices* 13, 98–101.

J Sparud [June 1993], "Fixing some space leaks without a garbage collector," in *Functional Programming Languages and Computer Architecture, Copenhagen*, ACM.

W Stoye [May 1985], "The implementation of functional languages using custom hardware," PhD Thesis, Computer Lab, University of Cambridge.

UNIX Programmer's Manual [Jan 1979] '**prof** command', section 1, Bell Laboratories, Murray Hill, N.J..

P Wadler [Sept 1987], "Fixing some space leaks with a garbage collector," *Software — Practice and Experience* 17, 595–608.

P Wadler & S Blott [Jan 1989], "How to make ad-hoc polymorphism less ad hoc," in *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*, ACM.

PL Wadler [June 1990], "Comprehending Monads," in *1990 ACM Conference on Lisp and Functional Programming*, Nice, France.

WM Waite [1973], "A sampling monitor for applications programs," *Software — Practice and Experience* 3, 75–79.

EP Wentworth [July 1990], "Pitfalls of conservative collection," *Software – Practice and Experience* 20, 719–727.

BA Wichmann [1973], *Algol 60 Compilation and Assessment*, Academic Press.

B Zorn [April 1992, revised August 1992], "The Measured Cost of Conservative Garbage Collection," Department of Computer Science, University of Colorado, Boulder Colorado, Technical Report C&U-C&S-573-92, Submitted for publication.

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

Appendix A

STG-machine Operational Semantics

This appendix presents STG-level operational semantics for the different profiling schemes, based on the STG language and the operational semantics presented in Peyton Jones [1992].

We first define the extended STG language used (Section A.1) and give an unprofiled operational semantics, expressed as a state transition system (Section A.2). This operational semantics is then extended to provide semantics for execution with lexical profiling (Section A.4), evaluation profiling (Section A.5), and our hybrid profiling scheme (Section A.6).

Though a complete syntax and semantics is given here, the accompanying discussion concentrates on the extensions required for profiling. The reader is refered to Peyton Jones [1992] for a detailed description of the standard STG language and its semantics.

A.1 The Extended STG Language

The STG language is an austere but recognisably-functional language. The language used here is the same as that presented in Peyton Jones [1992] with the following extensions.

- An scc expression form is introduced into the language which attaches a cost centre to an expression.

expr → ...	as before
| scc *cc expr*	Set Cost Centre

where *cc* is the cost centre label.

- Non-updateable closures (update flag \n) are classified into two distinct cases: those that are already in HNF (\r) and those that are not (\s). This distinction is required as they may have different cost semantics. The resulting update flags are:

  - \u — *Updateable.*

    Unevaluated closures that will be updated with their normal form.

  - \s — *Single-entry.*

    Unevaluated closures that promise that they will only be entered once. They are not updated with their normal form. It is up to the compiler to detect and label these single-entry closures (Launchbury et al. [1992]; Marlow [1993]).

  - \r — *Reentrant.*

    Closures that are already evaluated and may be entered (and re-evaluated) more than once. Manifest functions, constructors, and partial applications are always reentrant. They are never updated.

The complete extended STG language syntax is shown in figure A.1.

A.2 Unprofiled Operational Semantics

This section presents the operational semantics for the extended STG language executing without profiling. The semantics is presented using a state transition system. The state has six components:

1. the *code*, which takes one of several forms, given below;
2. the *argument stack*, *as*, which contains *values*;
3. the *return stack*, *rs*, which contains *continuations*;
4. the *update stack*, *us*, which contains *update frames*;

Program	$prog \rightarrow binds$		
Bindings	$binds \rightarrow var_1 = lf_1; \ldots; var_n = lf_n$	$n \geq 1$	
Lambda-forms	$lf \rightarrow vars_f \ \backslash \pi \ vars_a$ `->` $expr$		
Update flag	$\pi \rightarrow$ `u`	Updateable	
	| `s`	Single Entry	†
	| `r`	Reentrant	†
Expression	$expr \rightarrow$ `let` $binds$ `in` $expr$	Local definition	
	| `letrec` $binds$ `in` $expr$	Local recursion	
	| `case` $expr$ `of` $alts$	Case expression	
	| $var \ atoms$	Application	
	| $constr \ atoms$	Saturated constructor	
	| $prim \ atoms$	Saturated built-in op	
	| $literal$		
	| `scc` $cc \ expr$	Set Cost Centre	†
Alternatives	$alts \rightarrow aalt_1; \ldots; aalt_n; default$	$n \geq 0$ (Algebraic)	
	| $palt_1; \ldots; palt_n; default$	$n \geq 0$ (Primitive)	
Algebraic alt	$aalt \rightarrow constr \ vars$ `->` $expr$		
Primitive alt	$palt \rightarrow literal$ `->` $expr$		
Default alt	$default \rightarrow var$ `->` $expr$		
	| `default` `->` $expr$		
Literals	$literal \rightarrow$ `0#` | `1#` | ...	Primitive integers	
	| ...		
Primitive ops	$prim \rightarrow$ `+#` | `-#` | `*#` | `/#`	Primitive integer ops	
	| ...		
Variable lists	$vars \rightarrow \{var_1, \ldots, var_n\}$	$n \geq 0$	
Atom lists	$atoms \rightarrow \{atom_1, \ldots, atom_n\}$	$n \geq 0$	
	$atom \rightarrow var \mid literal$		

†Extensions made to standard STG language presented in Peyton Jones [1992].

Figure A.1: Syntax of the Extended STG language

5. the *heap*, h, which contains (only) *closures*;

6. the *global environment*, σ, which gives the addresses of all closures defined at top level.

Section A.3 introduces a seventh component, the *current cost centre* which is required for specifying the profiling semantics.

Sequences are used extensively in what follows. They are denoted using curly brackets, thus $\{a_1, \ldots, a_n\}$. The empty sequence is denoted $\{\}$; if as and bs are two sequences then $as +\!\!+ bs$ is their concatenation; and $a : as$ denotes the sequence obtained by adding the item a to the beginning of the sequence as. The length of a sequence as is denoted $length(as)$.

A *value* takes one of the following forms:

$Addr\ a$	A heap address
$Int\ n$	A primitive integer value

In the operational semantics, values are tagged with *Addr* and *Int* and so on to distinguish these different kinds of value (though the actual implementation avoids this). Further forms of value for other primitive data types, such as floating-point numbers, are handled exactly analogously to integers, so they are omitted to reduce clutter. Note that $w. w_1, \ldots$, is used to range over values, and ws to range over sequences of values.

The *heap*, h, is a mapping from *addresses*, ranged over by $a, a_1, \ldots$, to *closures*. Every closure is of the form

$$(vs\ \backslash\pi\ xs\ \texttt{->}\ e)\ ws$$

Intuitively, the lambda-form $(vs\ \backslash\pi\ xs\ \texttt{->}\ e)$ denotes the code of the closure, while the sequence of values ws gives the value of each of the free variables vs. (π is used to range over update flags, which can be either `u`, `s` or `r`.)

The *global environment* component of the state, σ, maps the name of each variable bound at the top level of the program to the address of its closure. These closures can all be allocated once and for all before execution begins. (Indeed, unlike the other components, σ does not change during execution.)

Finally, the *code* component of the state takes one of the following four forms, each of which is accompanied by its intuitive meaning:

Eval e ρ	Evaluate the expression e in environment ρ and apply its value to the arguments on the argument stack. The expression e is an arbitrarily complex STG-language expression.
Enter a	Apply the closure at address a to the arguments on the argument stack.
ReturnCon c ws	Return the constructor c applied to values ws to the continuation on the return stack.
ReturnInt k	Return the primitive integer k to the continuation on the return stack.

The *local environment* ρ, maps variable names to *values*. The notation $\rho[v \mapsto w]$ extends the map ρ with a mapping of the variable v to value w. This notation also extends in the obvious way to sequences of variables and values; for example $\rho[vs \mapsto ws]$.

The *val* function takes an atom (Figure A.1) and delivers a value:

$$
\begin{array}{lllll}
val & \rho & \sigma & k & = & Int\ k & \\
val & \rho & \sigma & v & = & \rho\ v & \text{if } v \in dom(\rho) \\
 & & & & = & \sigma\ v & \text{otherwise}
\end{array}
$$

If the atom is a literal k, *val* returns a primitive integer value. If it is a variable, *val* looks it up in ρ or σ as appropriate. *val* extends in the obvious way to sequences of variables: *val* $\rho\ \sigma\ vs$ is the sequence of values to which *val* $\rho\ \sigma$ maps the variables vs.

A.2.1 Initial State

First the initial state of the STG machine is specified. The general form of an STG program is as follows:

$$
\begin{array}{l}
g_1 \quad = vs_1\ \backslash \pi_1\ xs_1\ \texttt{->}\ e_1 \\
\ldots \\
g_n \quad = vs_n\ \backslash \pi_n\ xs_n\ \texttt{->}\ e_n
\end{array}
$$

One of the g_i will be `main`. Given this program, the corresponding initial state of the machine is:

Code	*Arg stack*	*Return stack*	*Update stack*	*Heap*	*Globals*
Eval (`main {}`) {}	{}	{}	{}	h_{init}	σ

$$\text{where } \sigma = \left[\begin{array}{l} g_1 \mapsto (Addr\ a_1) \\ \ldots \\ g_n \mapsto (Addr\ a_n) \end{array}\right] \qquad (0)$$

$$h_{init} = \left[\begin{array}{l} a_1 \mapsto (vs_1\ \backslash\pi_1\ xs_1\ \texttt{->}\ e_1)\ (\sigma\ vs_1) \\ \ldots \\ a_n \mapsto (vs_n\ \backslash\pi_n\ xs_n\ \texttt{->}\ e_n)\ (\sigma\ vs_n) \end{array}\right]$$

A.2.2 Applications

To perform a tail call, the values of the arguments are put on the argument stack, and the value of the function is entered.

	Eval $(f\ xs)\ \rho$		*as*	*rs*	*us*	h σ
such that	$val\ \rho\ \sigma\ f = Addr\ a$					
$\Longrightarrow$	*Enter* a	$(val\ \rho\ \sigma\ xs) +\!\!+ as$	*rs*	*us*	h	σ

(1)

The rule for entering a closure depends on the update flag. The rules for updatable closures (`\u` update flag) are given in Section A.2.6. For single entry (`\s`) and re-entrant (`\r`) closures the body is evaluated in an extended local environment.

	Enter a	*as*	*rs*	*us*	$h[a \mapsto (vs\ \backslash\texttt{r}\ xs\ \texttt{->}\ e)\ ws_f]$	σ
such that	$length(as) \geq length(xs)$					
$\Longrightarrow$	*Eval* $e\ \rho$	as'	*rs*	*us*	h	σ

where

$$\begin{array}{lcl} ws_a +\!\!+ as' & = & as \\ length(ws_a) & = & length(xs) \\ \rho & = & [vs \mapsto ws_f,\ xs \mapsto ws_a] \end{array}$$

(2)

The rule for single entry closures is identical in the unprofiled semantics.

Evaluating a constructor application simply moves into the *ReturnCon* state (see Section A.2.4):

$$\begin{array}{lllllll} & \mathit{Eval}\ (c\ xs)\ \rho & as & rs & us & h & \sigma \\ \Longrightarrow & \mathit{ReturnCon}\ c\ (\mathit{val}\ \rho\ \sigma\ xs) & as & rs & us & h & \sigma \end{array} \tag{3}$$

A.2.3 `let(rec)` Expressions

A `let` expression constructs one or more closures in the heap.

$$\begin{array}{lll} & \mathit{Eval} \left(\begin{array}{ll} \texttt{let} & x_1 = vs_1\ \backslash\pi_1\ xs_1\ \texttt{->}\ e_1 \\ & \cdots \\ & x_n = vs_n\ \backslash\pi_n\ xs_n\ \texttt{->}\ e_n \\ \texttt{in}\ e & \end{array} \right) \rho\ as\ rs\ us\ h\ \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho' \qquad as\ rs\ us\ h'\ \sigma \\ \text{where} & \rho' = \rho[x_1 \mapsto \mathit{Addr}\ a_1, \ldots, x_n \mapsto \mathit{Addr}\ a_n] \\ & h' = h \left[\begin{array}{l} a_1 \mapsto (vs_1\ \backslash\pi_1\ xs_1\ \texttt{->}\ e_1)\ (\rho_{rhs}\ vs_1) \\ \cdots \\ a_n \mapsto (vs_n\ \backslash\pi_n\ xs_n\ \texttt{->}\ e_n)\ (\rho_{rhs}\ vs_n) \end{array} \right] \\ & \rho_{rhs} = \rho \end{array} \tag{4}$$

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

A.2.4 Case Expressions and Data Constructors

The rule for `case` pushes a continuation onto the return stack and evaluates the case expression, e.

$$\begin{array}{lllllll} & \mathit{Eval}\ (\texttt{case}\ e\ \texttt{of}\ \mathit{alts})\ \rho & as & & rs & us & h & \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho & as & (\mathit{alts}, \rho) : rs & us & h & \sigma \end{array} \tag{5}$$

When the case expression, e, is evaluated and the result returned, the continuation is popped from the return stack and the appropriate alternative evaluated. The return rules for constructors and literals use intermediate return states *ReturnCon* and *ReturnInt* respectively. Primitive values are dealt with in the next section, while the rules for constructors are given next.

If the continuation on the return stack contains a pattern $c\ vs$ whose constructor c is the same as that being evaluated, the right-hand side of that alternative is evaluated, in the saved environment ρ augmented with bindings for the constructor fields, vs.

$$\begin{array}{lllllll} & \mathit{ReturnCon}\ c\ ws & as & (\ldots;\ c\ vs\ \texttt{->}\ e;\ \ldots,\ \rho) : rs & us & h & \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho[vs \mapsto ws] & as & & rs & us & h & \sigma \end{array} \tag{6}$$

If there is no such alternative, the default alternative is taken. The rule for this is easy when no variable is bound in the default case:

$$\begin{array}{llll} & \mathit{ReturnCon}\ c\ ws & as & \left(\begin{array}{l} c_1\ vs_1\ \texttt{->}\ e_1\texttt{;} \\ \ldots\texttt{;} \\ c_n\ vs_n\ \texttt{->}\ e_n\texttt{;} \\ \texttt{default ->}\ e_d \end{array}\right), \rho : rs\ \ us\ \ h\ \ \sigma \\ \text{such that} & c \neq c_i \quad (1 \leq i \leq n) & & \\ \Longrightarrow & \mathit{Eval}\ e_d\ \rho & as & rs\ \ us\ \ h\ \ \sigma \end{array} \tag{7}$$

The case where a variable is bound to the default is avoided for algebraic `case` expressions, as these would require the constructor to be allocated in the heap, by enforcing the following program transformation.

$$\texttt{case}\ e\ \texttt{of}\ \ldots\texttt{;}\ v\ \texttt{->}\ b \Longrightarrow \begin{array}{l} \texttt{let}\ v\ \texttt{=}\ xs\ \texttt{\textbackslash u \{\}->}\ e \\ \texttt{in} \\ \texttt{case}\ v\ \texttt{of}\ \ldots\texttt{;}\ \texttt{default ->}\ b \end{array}$$

Lastly, if there is no match and no default alternative, no rule matches, which is interpreted as failure.

A.2.5 Built-in Operations

The rule for evaluating a primitive literal, k, enters the *ReturnInt* state:

$$\begin{array}{lllllll} & \mathit{Eval}\ k\ \rho & as & rs & us & h & \sigma \\ \Longrightarrow & \mathit{ReturnInt}\ k & as & rs & us & h & \sigma \end{array} \tag{8}$$

A similar rule deals with the case where a variable bound to a primitive value is entered:

$$\begin{array}{llllllll} & \mathit{Eval}\ (f\ \{\})\ \rho[f \mapsto \mathit{Int}\ k] & as & rs & us & h & \sigma \\ \Longrightarrow & \mathit{ReturnInt}\ k & as & rs & us & h & \sigma \end{array} \tag{9}$$

The *ReturnInt* state looks for a continuation on the return stack chooses the appropriate alternative. First the case when there is a matching alternative.

$$\begin{array}{llllllll} & \mathit{ReturnInt}\ k & as & (\ldots\texttt{;}\ k\texttt{->}e\texttt{;}\ldots, \rho) : rs & us & h & \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho & as & rs & us & h & \sigma \end{array} \tag{10}$$

Next, the cases where the default alternative is taken:

$$\begin{array}{llll} & \mathit{ReturnInt}\ k & \mathit{as} & \left(\begin{array}{l} k_1\ \texttt{->}\ e_1\,; \\ \ldots\,; \\ k_n\ \texttt{->}\ e_n\,; \\ x\ \texttt{->}\ e \end{array}\right), \rho : rs\ \ us\ \ h\ \ \sigma \\ \text{such that} & k \neq k_i \quad (1 \leq i \leq n) & & \\ \Longrightarrow & \mathit{Eval}\ e\ \rho[x \mapsto \mathit{Int}\ k]\ \ \mathit{as} & & rs\ \ us\ \ h\ \ \sigma \end{array} \tag{11}$$

$$\begin{array}{llll} & \mathit{ReturnInt}\ k & \mathit{as} & \left(\begin{array}{l} k_1\ \texttt{->}\ e_1\,; \\ \ldots\,; \\ k_n\ \texttt{->}\ e_n\,; \\ \texttt{default}\ \texttt{->}\ e \end{array}\right), \rho : rs\ \ us\ \ h\ \ \sigma \\ \text{such that} & k \neq k_i \quad (1 \leq i \leq n) & & \\ \Longrightarrow & \mathit{Eval}\ e\ \rho & \mathit{as} & rs\ \ us\ \ h\ \ \sigma \end{array} \tag{12}$$

Finally, there are a family of rules for built-in arithmetic operations which, for each binary built-in operation $\oplus$, have the form:

$$\begin{array}{ll} & \mathit{Eval}\ (\oplus\ \{x_1, x_2\})\ \rho[x_1 \mapsto \mathit{Int}\ i_1,\ x_2 \mapsto \mathit{Int}\ i_2]\ \ \mathit{as}\ \ rs\ \ us\ \ h\ \ \sigma \\ \Longrightarrow & \mathit{ReturnInt}\ (i_1 \oplus i_2)\ \ \mathit{as}\ \ rs\ \ us\ \ h\ \ \sigma \end{array} \tag{13}$$

A.2.6 Updating Closures

When an updateable closure is entered, it pushes an *update frame* onto the update stack and makes the argument and return stacks empty. An update frame is a triple consisting of the previous argument stack, the previous return stack, and a pointer to the closure being entered. This closure will be updated with the result of evaluating the expression.

$$\begin{array}{llll} & \mathit{Enter}\ a & \mathit{as}\ \ rs & us\ \ h[a \mapsto (vs\ \texttt{\textbackslash u}\ \texttt{\{\}}\ \texttt{->}\ e)\ ws_f]\ \ \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho & \{\}\ \ \{\}\ \ (as, rs, a) : us & h\ \ \sigma \\ \text{where} & \rho = [vs \mapsto ws_f] & & \end{array} \tag{14}$$

When evaluation of the closure is complete an update is triggered. This can happen in one of two ways.

If the value of the closure is a data constructor, an attempt will be made to pop a continuation from the return stack, which will fail because the return stack is empty.

This failure triggers an update which updates the closure pointed to by the update frame, restores the argument and return stacks from the update frame, and tries again. It may be that the return stack is still empty requiring further updates to expose the continuation.

$$
\begin{array}{llllllll}
 & \mathit{ReturnCon}\ c\ ws & \{\} & \{\} & (as_u, rs_u, a_u) : us & h & \sigma \\
\Longrightarrow & \mathit{ReturnCon}\ c\ ws & as_u & rs_u & us & h_u & \sigma
\end{array}
$$

where vs is a sequence of arbitrary distinct variables
$length(vs) = length(ws)$
$h_u = h[a_u \mapsto (vs\ \backslash \mathtt{n}\ \{\}\ \mathtt{->}\ c\ vs)\ ws]$

(15)

If the value of the closure is a function, the function will attempt to bind arguments that are not present on the argument stack (because they were squirrelled away in the update frame). This failure to find enough arguments triggers an update.

$$
\begin{array}{llllll}
 & \mathit{Enter}\ a & as & \{\} & (as_u, rs_u, a_u) : us & h \quad \sigma \\
\text{such that} & h\ a = (vs\ \backslash \mathtt{r}\ xs\ \mathtt{->}\ e)\ ws_f & & & & \\
 & length(as) < length(xs) & & & & \\
\Longrightarrow & \mathit{Enter}\ a & as +\!\!+ as_u & rs_u & us & h_u \quad \sigma
\end{array}
$$

where $xs_1 +\!\!+ xs_2 = xs$
$length(xs_1) = length(as)$
f is an arbitrary variable
$h_u = h[a_u \mapsto ((f : xs_1)\ \backslash \mathtt{r}\ \{\}\ \mathtt{->}\ f\ xs_1)\ (a : as)]$

(16)

The closure to be updated (address a_u) is updated with a partial application of a to the arguments currently on the stack, as. Partial applications use of a fixed piece of code which unpacks the function and arguments stored in the closure if subsequently entered.

A.2.7 scc Expressions

Finally the semantics for an scc expression. During unprofiled execution an scc expression is simply ignored and the body evaluated.

$$
\begin{array}{llllllll}
 & \mathit{Eval}\ (\mathtt{scc}\ cc_{scc}\ e)\ \rho & as & rs & us & h & \sigma \\
\Longrightarrow & \mathit{Eval}\ e\ \rho & as & rs & us & h & \sigma
\end{array}
$$

(17)

A.3 Extending the Semantics for Profiling

The operational semantics of Section A.2 are now extended to include the manipulation of cost centres. As the lexical, evaluation and hybrid profiling schemes have different cost semantics separate STG-level operational semantics are presented in Sections A.4, A.5 and A.6 respectively.

These profiled STG-level semantics should only be read once the reader has a thorough understanding of the corresponding abstract cost semantics presented in Section 5.5.

All the STG-level profiling semantics require the following extensions to the state transition system presented in Section A.2:

- The current cost centre, *cc*, is added as an extra element to the machine state.
- All heap closures have the current cost centre attached to them when they are allocated (Section A.3.2). This is indicated by prefixing the heap object with the cost centre.
- The initial state is extended to include an initial cost centre and cost centres for all the top-level closures (Section A.3.1).

The extended transition systems concentrate on the rules that manipulate cost centres. The rules that do not manipulate cost centres are omitted for brevity. Where appropriate the rule numbers used correspond directly with those in Section A.2.

A.3.1 Initial State

In the initial state we must attach cost centres to the global or top-level declarations. The cost centre attached depends on the type of declaration.

- **Functions:**

  The cost of evaluating top-level functions are *subsumed* (see Section 4.1.4). This is indicated by attaching a special "SUB" cost centre. This is only a dummy cost centre — it is never assigned to the current cost centre.

- **Thunks:**

  Section 4.1.7 required all CAFs to be annoated with a cost centre. For the purpose of these semantics we introduce a single "CAF" cost centre which is attached to all CAF

closures. The costs incurred evaluating each CAF can be distinguished by attaching distinct cost centres.

- **Data Values:**

  Top-level data values are already evaluated. They are treated as thunks which happen to require no further evaluation — they simply return their value. The special cost centre "DATA" is attached. These are not built in the heap but may be created in the heap if a closure is updated with a copy of the data value.

We also initialise the current cost centre to the special cost centre "MAIN". This results in the cost of evaluating main being attributed to "MAIN".

Code	*Arg stack*	*Return stack*	*Update stack*	*Cost Centre*	*Heap*	*Globals*
Eval (main {}) {}	{}	{}	{}	MAIN	h_{init}	σ

where

$$\sigma = \begin{bmatrix} g_1 \mapsto (Addr\ a_1) \\ \ldots \\ g_n \mapsto (Addr\ a_n) \end{bmatrix}$$

$$h_{init} = \begin{bmatrix} a_1 \mapsto cc_1\ (vs_1\ \backslash \pi_1\ xs_1\ \texttt{->}\ e_1)\ (\sigma\ vs_1) \\ \ldots \\ a_n \mapsto cc_n\ (vs_n\ \backslash \pi_n\ xs_n\ \texttt{->}\ e_n)\ (\sigma\ vs_n) \end{bmatrix}$$

$$cc_i = \textit{if } length(xs_i) \geq 1 \textit{ then } \texttt{"SUB"} \textit{ else if } \pi_i = \texttt{r} \textit{ then } \texttt{"DATA"} \textit{ else } \texttt{"CAF"}$$

(0_{prof})

A.3.2 Constructing Heap Objects

Objects constructed in the heap by let(rec) expressions have the current cost centre attached to them.

$$Eval \left(\begin{array}{l} \texttt{let}\ \ x_1\ \texttt{=}\ vs_1\ \backslash \pi_1\ xs_1\ \texttt{->}\ e_1 \\ \quad \ldots \\ \quad\ \ x_n\ \texttt{=}\ vs_n\ \backslash \pi_n\ xs_n\ \texttt{->}\ e_n \\ \texttt{in}\ e \end{array} \right) \rho\ \ as\ \ rs\ \ us\ \ cc\ \ h\ \ \sigma$$

$$\Longrightarrow \quad Eval\ e\ \rho' \qquad .\qquad as\ \ rs\ \ us\ \ cc\ \ h'\ \ \sigma$$

where

$$\rho' = \rho\ [x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$$

$$h' = h \begin{bmatrix} a_1 \mapsto cc\ (vs_1\ \backslash \pi_1\ xs_1\ \texttt{->}\ e_1)\ (\rho_{rhs}\ vs_1) \\ \ldots \\ a_n \mapsto cc\ (vs_n\ \backslash \pi_n\ xs_n\ \texttt{->}\ e_n)\ (\rho_{rhs}\ vs_n) \end{bmatrix}$$

$$\rho_{rhs} = \rho$$

(4_{prof})

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

A.4 Lexical Profiling

Lexical scoping makes use of the tail call mechanism and return stack. The cost centre of the closure entered is loaded during a tail call. Return stack frames are used to save the cost centre when a non tail-call closure is entered. This cost centre is restored when evaluation returns.

This STG-machine implementation is based on the abstract push-enter rules in Figure 5.9.

A.4.1 Entering Closures

When a closure is entered the current cost centre is loaded with the cost centre stored in the closure, unless the closure is a top-level subsumed function.

We assume that the boxing transformation of Section 4.2.4 has already been applied. This ensures that any top-level functions which are passed as arguments have the cost centre of the referencing scope attached.

First the rule for top-level subsumed functions which does not load the current cost centre. These top-level functions are always re-entrant (`\r` update flag).

$$
\begin{array}{lll}
 & \mathit{Enter}\ a\ \ as\ \ rs\ \ us\ \ cc\ \ h\ \ \sigma & \\
 & & \\
\text{such that} & h\ a = \texttt{"SUB"}\ (vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & \\
 & \mathit{length}(as) \geq \mathit{length}(xs) & \\
 & & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho\ \ as'\ \ rs\ \ us\ \ cc\ \ h\ \ \sigma & \\
 & & \\
\text{where} & ws_a +\!\!+ as' = as & \\
 & \mathit{length}(ws_a) = \mathit{length}(xs) & \\
 & \rho = [vs \mapsto ws_f,\ xs \mapsto ws_a] &
\end{array}
\qquad (2_l^a)
$$

In all other cases the current cost centre is loaded with the cost centre of the entered closure cc_{enter}. This includes the entry of data closures which will simply return the data value.

$$
\begin{array}{lll}
 & \mathit{Enter}\ a \quad as \quad rs \quad us \quad cc & h \quad \sigma \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & \\
 & length(as) \geq length(xs) & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho \quad as' \quad rs \quad us \quad cc_{enter} & h \quad \sigma \\
\text{where} & ws_a +\!\!+ as' = as & \\
 & length(ws_a) = length(xs) & \\
 & \rho = [vs \mapsto ws_f,\ xs \mapsto ws_a] &
\end{array}
\qquad (2_l^b)
$$

The rule for single entry closures (\s update flag) is identical to rule 2_l^b. We discuss updateable closures (\u update flag) in Section A.4.3.

Note that the top-level subsumed functions are easily identified at compile time so no runtime test to is required to determine if the cost centre of the entered closure should be loaded. We generate entry code that "knows" if the cost centre should be loaded.

A.4.2 Saving and Restoring Cost Centres

When a case expression is evaluated the current cost centre must be saved so that it can be restored when the evaluation returns to the appropriate alternative.

$$
\begin{array}{lll}
 & \mathit{Eval}\ (\texttt{case}\ e\ \texttt{of}\ alts)\ \rho \quad as & rs \quad us \quad cc \quad h \quad \sigma \\
\Longrightarrow & \mathit{Eval}\ e\ \rho \quad as \quad (alts,\ cc,\ \rho) : rs & us \quad cc \quad h \quad \sigma
\end{array}
\qquad (5_l)
$$

This current cost centre is restored when evaluation returns.

$$
\begin{array}{lll}
 & \mathit{ReturnCon}\ c\ ws \quad as \quad (alts,\ cc_{ret},\ \rho) : rs & us \quad cc \quad h \quad \sigma \\
\text{such that} & alts = \ldots;\ c\ vs\ \texttt{->}\ e; \ldots & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho[vs \mapsto ws] \quad as & rs \quad us \quad cc_{ret} \quad h \quad \sigma
\end{array}
\qquad (6_l)
$$

The other return transition rules 7, 10, 11, and 12 restore the current cost centre in a similar fashion.

A.4.3 Updating Closures

When an updateable closure (\u update flag) is entered an update frame is pushed on the update stack (as for the unprofiled semantics) and the cost centre loaded from the closure.

$$
\begin{array}{lllllll}
 & \textit{Enter } a & as & rs & us & cc & h \;\; \sigma \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \backslash u\ \{\}\ \text{->}\ e)\ ws_f & & & & & \\
\Longrightarrow & \textit{Eval } e\ \rho\ \{\}\ \{\}\ (as, rs, a) : us & & & cc_{enter} & h & \sigma \\
\text{where} & \rho = \rho_{init}[vs \mapsto ws_f] & & & & &
\end{array}
\qquad (14_l)
$$

When a closure is updated a logically new heap closure[1] is built which contains the cost centre that evaluated the closure i.e. the current cost centre. There are two distinct cases: data value updates and partial application updates.

When a constructor sees an empty return stack the update stack is popped and the closure is updated with the data value. The current cost centre is attached to the updated closure. Any copies of the data closure that are generated by the update mechanism will have the same cost centre as the original.

$$
\begin{array}{lllllll}
 & \textit{ReturnCon } c\ ws & \{\} & \{\} & (as_u, rs_u, a_u) : us & cc & h \;\; \sigma \\
\Longrightarrow & \textit{ReturnCon } c\ ws & as_u & rs_u & us & cc & h_u \;\; \sigma \\
\text{where} & vs \text{ is a sequence of arbitrary distinct variables} & & & & & \\
 & length(vs) = length(ws) & & & & & \\
 & h_u = h[a_u \mapsto cc\ (vs\ \backslash r\ \{\}\ \text{->}\ c\ vs)\ ws] & & & & &
\end{array}
\qquad (15_l)
$$

When a λ-abstraction does not have enough arguments on the stack the closure is updated with a partial application. The cost centre of the function being entered is attached to the partial application (unless this is a "SUB" cost in which case the current cost centre is attached). This cost centre will be loaded if the partial application is ever entered.

[1] The use of indirections may avoid the actual construction of these new heap closures.

$$
\begin{array}{lll}
 & \textit{Enter } a & \textit{as}\ \ \{\}\ \ (as_u, rs_u, a_u) : us\ \ cc\ \ h\ \ \sigma \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & \\
 & length(as) < length(xs) & \\
\Longrightarrow & \textit{Enter } a\ \ as +\!\!+ as_u\ \ rs_u & us\ \ cc\ \ h_u\ \ \sigma \\
\text{where} & xs_1 +\!\!+ xs_2 = xs & \\
 & length(xs_1) = length(as) & \\
 & f \text{ is an arbitrary variable} & \\
 & h_u = h[a_u \mapsto cc_{pap}\ ((f : xs_1)\ \texttt{\textbackslash r}\ \{\}\ \texttt{->}\ f\ xs_1)\ (a : as)] & \\
 & cc_{pap} = \textsc{sub}(cc_{enter}, cc) & \\
 & \textsc{sub}(\texttt{"SUB"}, cc) = cc & \\
 & \textsc{sub}(cc_{enter}, cc) = cc_{enter} &
\end{array}
\qquad (16_l)
$$

To avoid a runtime test checking for a subsumed cost centre we specialise this rule into two cases, generating code that "knows" when a top-level subsumed function is being entered.

A.4.4 scc Expressions

Evaluating an scc expression under lexical profiling simply loads the current cost centre with the cost centre of the scc annotation, cc_{scc}. As this is a tail call the scc does not need to restore the cost centre when evaluation completes.

$$
\begin{array}{lllllll}
 & \textit{Eval } (\texttt{scc}\ cc_{scc}\ e)\ \rho & as & rs & us & cc & h\ \ \sigma \\
\Longrightarrow & \textit{Eval } e\ \rho & \{\} & \{\} & us & cc_{scc} & h\ \ \sigma
\end{array}
\qquad (17_l)
$$

A.5 Evaluation Profiling

Evaluation scoping makes use of the update frames to save and restore cost centres when closures are entered. The update frames are augmented with the cost centre to be restored once the closure evaluated and the update has been performed. A second form of update frame is introduced which is used to keep track of cost centres when no update is actually required. It does not contain a closure to update, just a cost centre to restore.

The STG-level evaluation semantics presented here are based on the our original implementation of evaluation profiling which only uses the update frames to save and restore cost centres (see Section 5.5.6). The STG-level implementation for the abstract push-enter rules in Figure 5.11 can be derived form the hybrid STG-level description in Appendix A.6 (see A.6.5).

A.5.1 Entering Closures and Saving Cost Centres

Entering an evaluated closure (\r update flag) does not modify the current cost centre. The (small) cost of entering these closures to extract the value within will be attributed to the cost centre demanding the value.

$$
\begin{array}{lll}
 & \mathit{Enter}\ a & as\ rs\ us\ cc\ h\ \sigma \\
 & & \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & \\
 & \mathit{length}(as) \geq \mathit{length}(xs) & \\
 & & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho\ as'\ rs\ us\ cc\ h\ \sigma & \\
 & & \\
\text{where} & ws_a +\!\!+ as' = as & \\
 & \mathit{length}(ws_a) = \mathit{length}(xs) & \\
 & \rho = [vs \mapsto ws_f,\ xs \mapsto ws_a] &
\end{array}
\qquad (2_e)
$$

This is crucial when entering functions as these do not return their value but evaluate the function as applied to the arguments on the stack. Under evaluation scoping this evaluation should be attributed to the cost centre of the application site not the cost centre attached to the function being entered. This is the fundamental distinction between evaluation and lexical scoping. Lexical scoping requires the loading of the cost centre on entry to a function (except for top-level subsumed functions) so that evaluation of the function body is attributed to the declaration site not the application site (see Section A.4.1).

When entering unevaluated closures, or thunks, the current cost centre is loaded with the thunk's cost centre. The demanding cost centre is saved and restored once evaluation of the closure has completed. For updateable closures (\u update flag) the demanding cost centre is added to the update frame. It will be restored when evaluation is complete and the closure updated.

$$
\begin{array}{lll}
 & \mathit{Enter}\ a & as\ rs\ us\ cc\ h\ \sigma \\
 & & \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash u}\ \texttt{\{\}}\ \texttt{->}\ e)\ ws_f & \\
 & & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho\ \{\}\ \{\}\ (as, rs, cc, a) : us\ cc_{enter}\ h\ \sigma & \\
 & & \\
\text{where} & \rho = \rho_{init}[vs \mapsto ws_f] &
\end{array}
\qquad (14^a_e)
$$

Single-entry closures (\s update flag) push a dummy update frame that just restores the cost centre when evaluation is complete and the update is triggered. No update will

actually be performed.

$$
\begin{array}{lllllll}
 & \textit{Enter } a & as & rs & us & cc & h \;\; \sigma \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash s}\ \{\}\ \texttt{->}\ e)\ ws_f & & & & & \\
\Longrightarrow & \textit{Eval } e\ \rho\ \{\}\ \{\}\ (as, rs, cc) : us & & & cc_{enter} & h & \sigma \\
\text{where} & \rho = \rho_{init}[vs \mapsto ws_f] & & & & &
\end{array}
\qquad (14^b_e)
$$

A.5.2 Updating Closures and Restoring Cost Centres

Finally the rules for updating. These fall into two categories:

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

First the rules for full updates. When a constructor sees an empty return stack an update is triggered. The updated closure requires the cost centre that evaluated and returned the constructor to be attached to it. Unfortunately this is below the level of detail of this semantics. We have not deemed it necessary to incorporate the required detail into the semantics as it is not critical to the semantics. The cost centre attached to updated closure has no affect on further evaluation — it only affects the attribution of heap allocation. The transition rule here refers to a "magic" value cc_{con}. Its value is simply the cost centre of the constructor that is being returned.

$$
\begin{array}{lllllllll}
 & \textit{ReturnCon } c\ ws & \{\} & \{\} & (as_u, rs_u, cc_u, a_u) : us & cc & h & \sigma \\
\Longrightarrow & \textit{ReturnCon } c\ ws & as_u & rs_u & us & cc_u & h_u & \sigma \\
\text{where} & vs \text{ is a sequence of arbitrary distinct variables} & & & & & & \\
 & length(vs) = length(ws) & & & & & & \\
 & h_u = h[a_u \mapsto cc_{con}\ (vs\ \texttt{\textbackslash r}\ \{\}\ \texttt{->}\ c\ vs)\ ws] & & & & & &
\end{array}
\qquad (15^a_e)
$$

When a λ-abstraction does not have enough arguments on the stack the closure is updated with a partial application. The cost centre of the function being entered is

attached to the partial application (unless this is a "SUB" cost in which case the current cost centre is attached). This cost centre only affects the heap profile — it has no effect on further evaluation as the costs of the application will be attributed to the application site.

$$
\begin{array}{lll}
 & \mathit{Enter}\ a & \mathit{as}\ \ \{\}\ \ (\mathit{as}_u, \mathit{rs}_u, \mathit{cc}_u, a_u) : \mathit{us}\ \ \mathit{cc}\ \ h\ \ \sigma \\
\text{such that} & h\ a = \mathit{cc}_{enter}\ (\mathit{vs}\ \texttt{\textbackslash r}\ \mathit{xs}\ \texttt{->}\ e)\ \mathit{ws}_f & \\
 & \mathit{length}(\mathit{as}) < \mathit{length}(\mathit{xs}) & \\
\Longrightarrow & \mathit{Enter}\ a\ \ \mathit{as} +\!\!+ \mathit{as}_u\ \ \mathit{rs}_u & \mathit{us}\ \ \mathit{cc}_u\ \ h_u\ \ \sigma \\
\text{where} & \mathit{xs}_1 +\!\!+ \mathit{xs}_2 = \mathit{xs} & \\
 & \mathit{length}(\mathit{xs}_1) = \mathit{length}(\mathit{as}) & \\
 & f \text{ is an arbitrary variable} & \\
 & h_u = h[a_u \mapsto \mathit{cc}_{pap}\ ((f : \mathit{xs}_1)\ \texttt{\textbackslash r}\ \{\}\ \texttt{->}\ f\ \mathit{xs}_1)\ (a : \mathit{as})] & \\
 & \mathit{cc}_{pap} = \mathrm{SUB}(\mathit{cc}_{enter}, \mathit{cc}) & \\
 & \mathrm{SUB}(\texttt{"SUB"}, \mathit{cc}) = \mathit{cc} & \\
 & \mathrm{SUB}(\mathit{cc}_{enter}, \mathit{cc}) = \mathit{cc}_{enter} &
\end{array}
\qquad (16^a_e)
$$

As for lexical scoping we specialise this rule into two cases, generating code that "knows" when a top-level subsumed function is being entered.

There is a corresponding pair of rules for dummy cost-centre updates. Here, the update frame only contains the cost centre to be restored, cc_u. There is no closure to be updated so the heap is left unchanged.

$$
\begin{array}{lll}
 & \mathit{ReturnCon}\ c\ \mathit{ws}\ \ \{\}\ \ \{\}\ \ (\mathit{as}_u, \mathit{rs}_u, \mathit{cc}_u) : \mathit{us}\ \ \mathit{cc}\ \ h\ \ \sigma \\
\Longrightarrow & \mathit{ReturnCon}\ c\ \mathit{ws}\ \ \mathit{as}_u\ \ \mathit{rs}_u & \mathit{us}\ \ \mathit{cc}_u\ \ h\ \ \sigma
\end{array}
\qquad (15^b_e)
$$

$$
\begin{array}{lll}
 & \mathit{Enter}\ a & \mathit{as}\ \ \{\}\ \ (\mathit{as}_u, \mathit{rs}_u, \mathit{cc}_u) : \mathit{us}\ \ \mathit{cc}\ \ h\ \ \sigma \\
\text{such that} & h\ a = \mathit{cc}_{enter}\ (\mathit{vs}\ \texttt{\textbackslash r}\ \mathit{xs}\ \texttt{->}\ e)\ \mathit{ws}_f & \\
 & \mathit{length}(\mathit{as}) < \mathit{length}(\mathit{xs}) & \\
\Longrightarrow & \mathit{Enter}\ a\ \ \mathit{as} +\!\!+ \mathit{as}_u\ \ \mathit{rs}_u & \mathit{us}\ \ \mathit{cc}_u\ \ h\ \ \sigma
\end{array}
\qquad (16^b_e)
$$

A.5.3 scc Expressions

Evaluating an scc expression under evaluation profiling loads the current cost centre with the cost centre of the scc annotation, cc_{scc}. A dummy cost-centre update frame, containing

the enclosing cost centre, cc, is pushed onto the update stack. It will be restored on completion of the evaluation of the expression e when the update will be triggered.

$$\begin{array}{|lllllllll|} \hline & \mathit{Eval}\ (\mathtt{scc}\ cc_{scc}\ e)\ \rho & as & rs & & us & cc & h & \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho & \{\} & \{\} & (as, rs, cc) : us & & cc_{scc} & h & \sigma \\ \hline \end{array} \quad (17_e)$$

A.6 Hybrid Profiling

The implementation of the hybrid profiling scheme combines the mechanisms of the lexical and evaluation implementations described in the previous sections:

- The return stack is used to save the cost centre on entry to a `case` and restore it when evaluation returns and the appropriate alternative is evaluated.
- Update frames are used to save the cost centre, restoring it if the result is a partial application of a function declared within the scope of a CAF or dictionary cost centre. This ensures that the costs of applying these λ-abstractions are attributed to the application site, not the declaration site.

This STG-machine implementation is based on the abstract push-enter rules in Figure 5.12.

A.6.1 Entering Closures

When an evaluated closure (`\r` update flag) is entered the current cost centre is loaded with the cost centre stored in the closure, unless the closure is a top-level subsumed function or a function declared in the scope of a CAF or dictionary cost centre. We have three distinct cases:

- Entering a top-level subsumed function. Since these top-level functions are easily identified at compile time the correct code can be generated. No runtime test is required.
- The general case for entering a λ-abstraction requires a runtime test, captured by the EVAL selector, to check for λ-abstractions declared in the scope of a CAF or dictionary cost centre. If the scope declaring the λ-abstraction is known at compile time the appropriate code can be generated and the runtime test omitted.

As for lexical scoping, we assume that the boxing transformation of Section 4.2.4 has already been applied. This ensures that any top-level functions that are passed as arguments have the cost centre of the referencing scope attached.

- Entering an evaluated data closure (no arguments) always loads the cost centre. No runtime test is required.

The three rules are given below:

$$
\begin{array}{lll}
 & \textit{Enter } a \quad as \quad rs \quad us \quad cc \quad h \quad \sigma & \\
\text{such that} & h\ a = \texttt{"SUB"}\ (vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & \\
 & length(as) \geq length(xs) & \\
\Longrightarrow & \textit{Eval } e\ \rho \quad as' \quad rs \quad us \quad cc \quad h \quad \sigma & (2^a_h) \\
\text{where} & ws_a +\!\!+ as' = as & \\
 & length(ws_a) = length(xs) & \\
 & \rho = [vs \mapsto ws_f,\ xs \mapsto ws_a] &
\end{array}
$$

$$
\begin{array}{lll}
 & \textit{Enter } a \quad as \quad rs \quad us \quad cc \quad h \quad \sigma & \\
\text{such that} & h\ a = cc_a\ (vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & \\
 & length(xs) > 0 & \\
 & length(as) \geq length(xs) & \\
\Longrightarrow & \textit{Eval } e\ \rho \quad as' \quad rs \quad us \quad cc_{enter} \quad h \quad \sigma & (2^b_h) \\
\text{where} & ws_a +\!\!+ as' = as & \\
 & length(ws_a) = length(xs) & \\
 & \rho = [vs \mapsto ws_f,\ xs \mapsto ws_a] & \\
 & cc_{enter} = \textsc{eval}(cc_a, cc) & \\
 & \textsc{eval}(\texttt{"CAF"}, cc) = cc & \\
 & \textsc{eval}(\texttt{"DICT"}, cc) = cc & \\
 & \textsc{eval}(cc_a, cc) = cc_a &
\end{array}
$$

$$
\begin{array}{lll}
 & \textit{Enter } a \quad as \quad rs \quad us \quad cc \quad h \quad \sigma & \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash r}\ \{\}\ \texttt{->}\ e)\ ws_f & \\
\Longrightarrow & \textit{Eval } e\ \rho \quad as \quad rs \quad us \quad cc_{enter} \quad h \quad \sigma & (2^c_h) \\
\text{where} & \rho = [vs \mapsto ws_f] &
\end{array}
$$

The rule for entering single entry closures (\s update flag) requires a dummy cost-centre update frame. This is discussed along with updateable closures (\u update flag) in Section A.6.3.

A.6.2 Saving and Restoring Cost Centres

When a case expression is evaluated the current cost centre must be saved so that it can be restored when the evaluation returns to the appropriate alternative. This is identical to the save and restore mechanism used in the lexical implementation (Section A.4.2).

$$\begin{array}{lllllllll} & \mathit{Eval}\ (\texttt{case}\ e\ \texttt{of}\ \mathit{alts})\ \rho & \mathit{as} & & \mathit{rs} & \mathit{us} & \mathit{cc} & h & \sigma \\ \Longrightarrow & \mathit{Eval}\ e\ \rho & \mathit{as} & (\mathit{alts},\ \mathit{cc},\ \rho) : \mathit{rs} & \mathit{us} & \mathit{cc} & h & \sigma \end{array} \qquad (5_h)$$

This current cost centre is restored when evaluation returns.

$$\begin{array}{lllllllll} & \mathit{ReturnCon}\ c\ \mathit{ws} & \mathit{as} & (\mathit{alts},\ \mathit{cc}_{ret},\ \rho) : \mathit{rs} & \mathit{us} & \mathit{cc} & h & \sigma \\ \text{such that} & \mathit{alts} = \ldots ;\ c\ \mathit{vs}\ \texttt{->}\ e ; \ldots & & & & & & & \\ \Longrightarrow & \mathit{Eval}\ e\ \rho[\mathit{vs} \mapsto \mathit{ws}] & \mathit{as} & & \mathit{rs} & \mathit{us} & \mathit{cc}_{ret} & h & \sigma \end{array} \qquad (6_h)$$

The other return transition rules 7, 10, 11, and 12 restore the current cost centre in a similar fashion.

A.6.3 Updating closures

When entering unevaluated closures, or thunks, the current cost centre is loaded with the thunk's cost centre. The demanding cost centre is saved. This is restored when evaluation completes if the result is a λ-abstraction (i.e. partial application) declared in the scope of a CAF or dictionary cost centre. The hybrid update rules are similar to those for evaluation scoping (Sections A.5.1 and A.5.2) except that the cost centre is only restored when a CAF or dictionary PAP update occurs.

For updateable closures (\u update flag) the demanding cost centre is added to the update frame.

$$
\begin{array}{lll}
 & \mathit{Enter}\ a\ \ as\ \ rs & us\ \ cc\ \ \ h\ \ \sigma \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash u}\ \{\}\ \texttt{->}\ e)\ ws_f & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho\ \ \{\}\ \ \{\}\ \ (as, rs, cc, a) : us\ \ cc_{enter}\ \ h\ \ \sigma & \\
\text{where} & \rho = \rho_{init}[vs \mapsto ws_f] &
\end{array}
\qquad (14^a_h)
$$

Single-entry closures (`\s` update flag) push a dummy update frame which will just restore the cost centre if required when the update is triggered. No update will actually be performed.

$$
\begin{array}{lll}
 & \mathit{Enter}\ a\ \ as\ \ rs & us\ \ cc\ \ \ h\ \ \sigma \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash s}\ \{\}\ \texttt{->}\ e)\ ws_f & \\
\Longrightarrow & \mathit{Eval}\ e\ \rho\ \ \{\}\ \ \{\}\ \ (as, rs, cc) : us\ \ cc_{enter}\ \ h\ \ \sigma & \\
\text{where} & \rho = \rho_{init}[vs \mapsto ws_f] &
\end{array}
\qquad (14^b_h)
$$

The rule for updating with a constructor proceeds as for lexical scoping. The current cost centre is attached to the updated closure and the demanding cost centre is not restored.

$$
\begin{array}{lll}
 & \mathit{ReturnCon}\ c\ ws\ \ \{\}\ \ \{\}\ \ (as_u, rs_u, cc_u, a_u) : us\ \ cc\ \ h\ \ \sigma & \\
\Longrightarrow & \mathit{ReturnCon}\ c\ ws\ \ as_u\ \ rs_u\ \ us\ \ cc\ \ h_u\ \ \sigma & \\
\text{where} & vs \text{ is a sequence of arbitrary distinct variables} & \\
 & length(vs) = length(ws) & \\
 & h_u = h[a_u \mapsto cc\ (vs\ \texttt{\textbackslash r}\ \{\}\ \texttt{->}\ c\ vs)\ ws] &
\end{array}
\qquad (15^a_h)
$$

Dummy updates triggered by a returning constructor are simply removed as the cost centre does not need to be restored.

$$
\begin{array}{lll}
 & \mathit{ReturnCon}\ c\ ws\ \ \{\}\ \ \{\}\ \ (as_u, rs_u, cc_u) : us\ \ cc\ \ h\ \ \sigma & \\
\Longrightarrow & \mathit{ReturnCon}\ c\ ws\ \ as_u\ \ rs_u\ \ us\ \ cc\ \ h\ \ \sigma &
\end{array}
\qquad (15^b_h)
$$

Partial application updates restore the demanding cost centre only if the cost centre of the closure being entered is a CAF or dictionary cost centre. If the cost centre of the closure being entered is "SUB" then the decision is based on the value of the current

cost centre. This requires a runtime test, which is captured by the EVAL selector. (The SUB selector does not require a test as we generate code that "knows" when a top-level subsumed function is entered.)

$$
\begin{array}{lll}
 & \textit{Enter } a \quad as \quad \{\} \quad (as_u, rs_u, cc_u, a_u) : us \quad cc \quad h \quad \sigma & \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & \\
 & length(as) < length(xs) & \\
\Longrightarrow & \textit{Enter } a \quad as +\!\!+ as_u \quad rs_u \quad us \quad cc_{restore} \quad h_u \quad \sigma & \\
\text{where} & xs_1 +\!\!+ xs_2 = xs & \\
 & length(xs_1) = length(as) & \\
 & f \text{ is an arbitrary variable} & \\
 & h_u = h[a_u \mapsto cc_{pap}\ ((f : xs_1)\ \texttt{\textbackslash r}\ \{\}\ \texttt{->}\ f\ xs_1)\ (a : as)] & \\
 & cc_{pap} = \text{SUB}(cc_{enter}, cc) & \\
 & cc_{restore} = \text{EVAL}(cc_{pap}, cc_u) & \\
 & \text{SUB}(\texttt{"SUB"}, cc) = cc & \\
 & \text{SUB}(cc_{enter}, cc) = cc_{enter} & \\
 & \text{EVAL}(\texttt{"CAF"}\ , cc_u) = cc_u & \\
 & \text{EVAL}(\texttt{"DICT"}, cc_u) = cc_u & \\
 & \text{EVAL}(cc_{pap}\ , cc_u) = cc_{pap} &
\end{array}
\qquad (16^a_h)
$$

$$
\begin{array}{lll}
 & \textit{Enter } a \quad as \quad \{\} \quad (as_u, rs_u, cc_u) : us \quad cc \quad h \quad \sigma & \\
\text{such that} & h\ a = cc_{enter}\ (vs\ \texttt{\textbackslash r}\ xs\ \texttt{->}\ e)\ ws_f & \\
 & length(as) < length(xs) & \\
\Longrightarrow & \textit{Enter } a \quad as +\!\!+ as_u \quad rs_u \quad us \quad cc_{restore} \quad h \quad \sigma & \\
\text{where} & cc_{restore} = \text{EVAL}(\text{SUB}(cc_{enter}, cc), cc_u) & \\
 & \text{SUB}(\texttt{"SUB"}, cc) = cc & \\
 & \text{SUB}(cc_{enter}, cc) = cc_{enter} & \\
 & \text{EVAL}(\texttt{"CAF"}\ , cc_u) = cc_u & \\
 & \text{EVAL}(\texttt{"DICT"}, cc_u) = cc_u & \\
 & \text{EVAL}(cc_{pap}\ , cc_u) = cc_{pap} &
\end{array}
\qquad (16^b_h)
$$

The demanding cost centre is only restored if the result is λ-abstraction declared in the scope of a CAF or dictionary cost centre. If it is known at compile time that the demanding cost centre will not be restored (i.e. the result is a constructor or a λ-abstraction that is not declared in a CAF or dictionary cost centre) then the cost centre need not be saved

in the update frame. This optimisation is particularly significant for single entry closures as the dummy update frame can then be omitted altogether.

A.6.4 scc Expressions

Evaluating an scc expression under the hybrid profiling scheme loads the current cost centre with the cost centre of the scc annotation, cc_{scc}. A dummy cost-centre update frame, containing the enclosing cost centre, cc, is pushed onto the update stack. It will be restored on completion of the evaluation of the expression e when the (dummy) update is triggered if the result is a λ-abstraction declared in the scope of a CAF or dictionary.

	Eval (scc cc_{scc} e) ρ	*as*	*rs*		*us*	*cc*	*h*	σ
$\Longrightarrow$	*Eval* e ρ	{}	{}	$(as, rs, cc) : us$		cc_{scc}	*h*	σ

(17_e)

This is identical to the scc rule for evaluation scoping (Section A.5.3), except that the enclosing cost centre will only be restored if the result is a λ-abstraction with a CAF or dictionary cost centre.

A.6.5 Evaluation scoping

This STG-machine implementation of the evaluation scoping push-enter rules in Figure 5.11 can be derived from the hybrid implementation given above. All that is required is for the EVAL selector used in rules 16_h^b, 16_h^a and 2_h^b to always select/restore the cost centre of the application site i.e.

$$\text{EVAL}_e(cc_{fn}, cc_{ap}) = cc_{ap}$$

This is a known action which does not require any runtime test.

Appendix B

Profiling Documentation

The following sections are taken from the user documentation of the Glasgow Haskell compiler (Version 0.22).

B.1 Compiling programs for profiling

To make use of the cost centre profiling system *ALL* modules must be compiled and linked with the `-prof` profiling option.

There are a number of additional options which affect the cost centre declarations within particular Haskell modules. These do not have to be used consistently.

`-prof`: Compile module with cost centre profiling (based on the hybrid cost semantics). `scc` annotations in the Haskell source will be be recognised, causing the costs incurred by the enclosed expression to be attributed to the named cost centre. *ALL* modules must be compiled and linked with this option.

If the `-prof` option is not specified `scc` annotations in the source will be ignored. This allows you to compile the source normally after placing explicit `scc` annotations in your source.

`-auto`: All top-level, exported declarations are automatically annotated with cost centres. The label given to cost centre annotation is the name of the declaration. Explicit `scc` annotations are still recognised.

`-auto-all`: All top-level (included non-exported) declarations are automatically annotated with cost centres.

`-caf-all`: By default the costs of all CAFs in a module are attributed to a single CAF cost centre. This option requests that the costs of each CAF be attributed to its own cost centre.

`-dict-all`: By default the costs of all dictionaries in a module are attributed to a single DICT cost centre. This option requests that the dictionary costs of each instance be attributed to a separate cost centre.

`-ignore-scc`: Forces any `scc` annotations in the module's source to be ignored. This is a surprisingly useful option as it allows a module which has had `scc` annotations

added to be compiled for profiling with the annotations being ignored. There is no need to remove the annotations.

`-G<group>`: Specifies the ⟨group⟩ to be attached to all the cost centres declared in the module. If no group is specified it defaults to the module name.

Alternative profiling semantics may also be available. To use these the runtime system and prelude libraries must have been built for the alternative profiling setup. This is done using a particular UserWay setup. The alternative profiling system will normally be invoked using the options:

`-lex`: for lexical profiling.

`-eval`: for evaluation profiling.

All modules must be consistently compiled with the `-lex` or `-eval` option instead of the `-prof` option. The other profiling options are still applicable.

Finally we note that the options which dump the program source during compilation may be useful to determine exactly what code is being profiled. Useful options are:

`-ddump-ds`: dump after desugaring. Any automatic `scc` annotations will have been added.

`-ddump-simpl`: dump after simplification.

`-ddump-stg`: dump the STG-code immediately before code generation.

B.2 Controlling the profiler at runtime

These flags are passed to the runtime system between the usual `+RTS` and `-RTS` options. They will only have an effect if the program was compiled and linked with the `-prof` options (see Section B.1).

`-p<sort>` **or** `-P<sort>`: The `-p` option produces an aggregate profile report describing the amount of time and allocation consumed by each cost centre. The report is written into the file ⟨program⟩`.prof`.

The `-P` option produces a more detailed report containing the actual time and allocation data as well. It also produces *serial* time-profiling information, in the file ⟨program⟩`.time`. This can be converted into a (somewhat unsatisfactory) PostScript graph using `hp2ps` (see Section B.3.2). The profiling interval may be set using the `-i<secs>` option (the default is 1 second between data samples).

The ⟨sort⟩ indicates how the cost centres are to be sorted in the report. Valid ⟨sort⟩ options are:

`T`: time, largest first (the default);

`A`: bytes allocated, largest first;

`C`: alphabetically by group, module and label.

The `T` and `A` ⟨sort⟩s place all the CAF and dictionary cost centres at the end.

`-h<break-down>`: Produce a detailed serial heap profile of the space occupied by live closures at regular points in time over the run of the program. The profile is written to the file ⟨program⟩`.hp` from which a PostScript graph can be produced using `hp2ps` (see Section B.3.2).

The heap space profile may be broken down by different criteria:

`-hC`: cost centre which allocated the closure (the default).

`-hM`: cost centre module which allocated the closure.

`-hG`: cost centre group which allocated the closure.

`-hD`: closure description — a string describing the closure.

`-hY`: closure type — a string describing the closure's type.

`-hT<ints>,<start>`: the time interval the closure was created. `<ints>` specifies the no. of interval bands plotted (default 18) and `<start>` the number of seconds after which the reported intervals start (default 0.0).

By default all live closures in the heap are profiled, but particular closures of interest can be selected (see below).

The heap profiling interval may be set using the `-i<secs>` option (the default is 1 second between heap profile samples). This is used to adjust the number of sample points during the run of the program.

Finally we note that heap profiling uses hash tables. If these tables should fill the run will abort. The `-z<tbl><size>` option is used to increase the size of the relevant hash table (`C`, `M`, `G`, `D` or `Y`, defined as for ⟨break-down⟩ above). The actual size used is the next largest power of 2.

The heap profile can be restricted to particular closures of interest. The closures of interest can selected by the attached cost centre (module:label, module and group), closure category (description, type, and kind) and closure age using the following options:

`-c{<mod>:<lab>,<mod>:<lab>...}`: Selects individual cost centre(s).

`-m{<mod>,<mod>...}`: Selects all cost centres from the module(s) specified.

`-g{<grp>,<grp>...}`: Selects all cost centres from the groups(s) specified.

`-d{<des>,<des>...}`: Selects closures which have one of the specified descriptions.

`-y{<typ>,<typ>...}`: Selects closures which have one of the specified type descriptions.

`-k{<knd>,<knd>...}`: Selects closures which are of one of the specified closure kinds. Valid closure kinds are `CON` (constructor), `FN` (manifest function), `PAP` (partial application), `BH` (black hole) and `THK` (thunk).

`-a<age>`: Selects closures which have survived ⟨age⟩ complete intervals.

The space occupied by a closure will be reported in the heap profile if the closure satisfies the following logical expression:

([-c] or [-m] or [-g]) and ([-d] or [-y] or [-k]) and [-a]

where a particular option is true if the closure (or its attached cost centre) is selected by the option (or the option is not specified).

B.3 Graphical post-processors

Utility programs which produce graphical profiles.

B.3.1 stat2resid

USAGE: `stat2resid` [⟨file⟩[`.stat`] [⟨outfile⟩]]

The program `stat2resid` converts a detailed garbage collection statistics file produced by the `-S` runtime option into a PostScript heap residency graph. The garbage collection statistics file can be produced without compiling your program for profiling.

By convention, the file to be processed by `stat2resid` has a `.stat` extension. If the ⟨outfile⟩ is not specified the PostScript will be written to ⟨file⟩`.resid.ps`. If ⟨file⟩ is omitted entirely, then the program behaves as a filter.

The plot can not be produced from the statistics file for a generational collector, though a suitable stats file can be produced using the `-F2s` runtime option when the program has been compiled for generational garbage collection (the default).

`stat2resid` is distributed in `ghc/utils/stat2resid`.

B.3.2 hp2ps

USAGE: `hp2ps` [flags] [⟨file⟩[`.stat`]]

The program `hp2ps` converts a heap profile as produced by the `-h<break-down>` runtime option into a PostScript graph of the heap profile. By convention, the file to be processed by `hp2ps` has a `.hp` extension. The PostScript output is written to ⟨file⟩`.ps`. If ⟨file⟩ is omitted entirely, then the program behaves as a filter.

`hp2ps` is distributed in `ghc/utils/hp2ps`. It was originally developed by David Wakeling as part of the **hbc/lml** heap profiler. Various extensions have been made to the original program by Patrick Sansom.

The flags are:

`-d` In order to make graphs more readable, `hp2ps` sorts the shaded bands for each identifier. The default sort ordering is for the bands with the largest area to be stacked on top of the smaller ones. The `-d` option causes rougher bands (those representing series of values with the largest standard deviations) to be stacked on top of smoother ones.

`-i[+|-]` The `-i` option causes the bands to be sorted lexicographically by the identifier string. `+` indicates the greatest string will be on top (the default) and `-` indicates the least string will be on top. `-i+` is used to sort the creation-time heap profiles.

`-p` Use previous parameters. By default, the PostScript graph is automatically scaled both horizontally and vertically so that it fills the page. However, when preparing a series of graphs for use in a presentation, it is often useful to draw a new graph using the same scale, shading and ordering as a previous one. The `-p` flag causes the graph to be drawn using the parameters determined by a previous run of `hp2ps` on file. These are extracted from file`.aux`.

`-m<int>` Normally a profile is limited to 20 bands with additional identifiers being grouped into an `OTHER` band. The `-m` flag specifies an alternative band limit (the maximum is 20).

`-m0` requests the band limit to be removed. As many bands as necessary are produced. However no key is produced as it won't fit! It is useful for displaying creation time heap profiles with many bands.

`-t<float>` Normally trace elements which sum to a total of less than 1% of the profile are removed from the profile. The `-t` option allows this percentage to be modified (maximum 5%).

`-t0` requests no trace elements to be removed from the profile, ensuring that all the data will be displayed.

`-e<float>[in|mm|pt]` Generate encapsulated PostScript suitable for inclusion in LaTeX documents. Usually, the PostScript graph is drawn in landscape mode in an area 9 inches wide by 6 inches high, and `hp2ps` arranges for this area to be approximately centred on a sheet of a4 paper. This format is convenient of studying the graph in detail, but it is unsuitable for inclusion in LaTeX documents. The `-e` option causes the graph to be drawn in portrait mode, with float specifying the width in inches, millimetres or points (the default). The resulting PostScript file conforms to the Encapsulated Post Script (EPS) convention, and it can be included in a LaTeX document using Rokicki's dvi-to-PostScript converter `dvips`.

`-g` Create output suitable for the `gs` PostScript previewer (or similar). In this case the graph is printed in portrait mode without scaling. The output is unsuitable for a laser printer.

`-b` Normally, `hp2ps` puts the title of the graph in a small box at the top of the page. However, if the JOB string is too long to fit in a small box (more than 35 characters), then `hp2ps` will choose to use a big box instead. The `-b` option forces `hp2ps` to use a big box.

`-s` Use a small box for the title.

`-?` Print out usage information.

Appendix C

Clausify

This Appendix contains the Haskell source for **clausify**, after the improvements by Runciman & Wakeling [1993] have been incorporated. A brief description of the program can be found in Section 6.1.

C.1 Haskell Source

```
-- CLAUSIFY: Reducing Propositions to Clausal Form
-- Colin Runciman, University of York, 10/90
--
-- An excellent benchmark is: (a = a = a) = (a = a = a) = (a = a = a)
--
-- Optimised version: based on Runciman & Wakeling [1993]
-- Patrick Sansom, University of Glasgow, 2/94

module Main(main) where

-- the main program: reads stdin and writes stdout
main = readChan stdin exit ( \input ->
       appendChan stdout (clausify input) exit done)

-- convert lines of propostions input to clausal forms
clausify input = concat
                 (interleave (repeat "prop> ")
                             (map clausifyline (lines input)))

clausifyline = concat . map disp . clauses . parse
```

```
-- the main pipeline from propositional formulae to printed clauses
-- with explicit scc annotations (lex and eval scoping)
-- clauses = unicl . split . disin . negin . elim
clauses = (\x -> scc "unicl" unicl x) .
          (\x -> scc "split" split x) .
          (\x -> scc "disin" disin x) .
          (\x -> scc "negin" negin x) .
          (\x -> scc "elim"  elim  x)

data StackFrame = Ast Formula | Lex Char

data Formula = Sym Char
             | Not Formula
             | Dis Formula Formula
             | Con Formula Formula
             | Imp Formula Formula
             | Eqv Formula Formula

-- separate positive and negative literals, eliminating duplicates
clause p = clause' p ([] , [])
           where
           clause' (Dis p q)       x   = clause' p (clause' q x)
           clause' (Sym s)       (c,a) = (insert s c , a)
           clause' (Not (Sym s)) (c,a) = (c , insert s a)

-- shift disjunction within conjunction
disin (Con p q) = Con (disin p) (disin q)
disin (Dis p q) = disin' (disin p) (disin q)
disin p         = p

-- auxilary definition encoding (disin . Dis)
disin' (Con p q) r = Con (disin' p r) (disin' q r)
disin' p (Con q r) = Con (disin' p q) (disin' p r)
disin' p q         = Dis p q

-- format pair of lists of propositional symbols as clausal axiom
disp (l,r) = interleave l spaces ++ "<=" ++ interleave spaces r ++ "\n"

-- eliminate connectives other than not, disjunction and conjunction
elim (Sym s)    = Sym s
elim (Not p)    = Not (elim p)
elim (Dis p q)  = Dis (elim p) (elim q)
elim (Con p q)  = Con (elim p) (elim q)
elim (Imp p q)  = Dis (Not (elim p)) (elim q)
elim (Eqv f f') = Con (elim (Imp f f')) (elim (Imp f' f))
```

```
-- remove duplicates and any elements satisfying p
filterset p s = filterset' [] p s

filterset' res p []     = []
filterset' res p (x:xs) = if (notElem x res) && (p x) then
                              x : filterset' (x:res) p xs
                          else
                              filterset' res p xs

-- insertion of an item into an ordered list
insert x []     = [x]
insert x (y:ys) = if x < y then x:y:ys
                  else if x > y then y : insert x ys
                  else y:ys

interleave (x:xs) ys = x : interleave ys xs
interleave []     _  = []

-- shift negation to innermost positions
negin (Not (Not p))   = negin p
negin (Not (Con p q)) = Dis (negin (Not p)) (negin (Not q))
negin (Not (Dis p q)) = Con (negin (Not p)) (negin (Not q))
negin (Dis p q)       = Dis (negin p) (negin q)
negin (Con p q)       = Con (negin p) (negin q)
negin p = p

-- the priorities of symbols during parsing
opri '(' = 0
opri '=' = 1
opri '>' = 2
opri '|' = 3
opri '&' = 4
opri '~' = 5

-- parsing a propositional formula
parse t = f where [Ast f] = parse' t []

parse' []      s = redstar s
parse' (' ':t) s = parse' t s
parse' ('(':t) s = parse' t (Lex '(' : s)
parse' (')':t) s = parse' t (x:s')
                   where
                   (x : Lex '(' : s') = redstar s
parse' (c:t)   s = if inRange ('a','z') c then parse' t (Ast (Sym c) : s)
                   else if spri s > opri c then parse' (c:t) (red s)
                   else parse' t (Lex c : s)
```

```
-- reduction of the parse stack
red (Ast p : Lex '=' : Ast q : s) = Ast (Eqv q p) : s
red (Ast p : Lex '>' : Ast q : s) = Ast (Imp q p) : s
red (Ast p : Lex '|' : Ast q : s) = Ast (Dis q p) : s
red (Ast p : Lex '&' : Ast q : s) = Ast (Con q p) : s
red (Ast p : Lex '~' : s)         = Ast (Not p) : s

-- iterative reduction of the parse stack
redstar = while ((/=) 0 . spri) red

spaces = repeat ' '

-- split conjunctive proposition into a list of conjuncts
split p = split' p []
          where
          split' (Con p q) a = split' p (split' q a)
          split' p a = p : a

-- priority of the parse stack
spri (Ast x : Lex c : s) = opri c
spri s = 0

-- does any symbol appear in both consequent and antecedant of clause
tautclause (c,a) = [x | x <- c, x `elem` a] /= []

-- form unique clausal axioms excluding tautologies
unicl = filterset (not . tautclause) . map clause

-- functional while loop
while p f x = if p x then while p f (f x) else x

{- Runciman & Wakeling's Original Version Definitions:

conjunct (Con p q) = True
conjunct p         = False

disin (Dis p (Con q r)) = Con (disin (Dis p q)) (disin (Dis p r))
disin (Dis (Con p q) r) = Con (disin (Dis p r)) (disin (Dis q r))
disin (Dis p q) =
  if conjunct dp || conjunct dq then disin (Dis dp dq)
  else (Dis dp dq)
  where
  dp = disin p
  dq = disin q
disin (Con p q) = Con (disin p) (disin q)
disin p = p
```

```
unicl a = foldr unicl' [] a
          where
          unicl' p x = if tautclause cp then x else insert cp x
                       where
                       cp = clause p
-}
```

C.2 Unboxing Optimisation

Below are the modified declarations which use **ghc**'s unboxed characters (Peyton Jones & Launchbury [1991]). They must be compiled using the `-glasgow-exts` flag. Most of the modifications are defining operations on the user defined data type `Literals` which replaces the `List` of literals in the original program. It is not possible to construct a `List` of unboxed literals directly since an unboxed value cannot be passed to polymorphic function.

```
data Formula = Sym Char#
             | Not Formula
             | Dis Formula Formula
             | Con Formula Formula
             | Imp Formula Formula
             | Eqv Formula Formula

data Literals = End | Lit Char# Literals

clause p = clause' p (End, End)
           where
           clause' (Dis p q)       x   = clause' p (clause' q x)
           clause' (Sym s)       (c,a) = (insert s c , a)
           clause' (Not (Sym s)) (c,a) = (c , insert s a)

disp (l,r) = interleave (unlit l) spaces ++ "<=" ++
                 interleave spaces (unlit r) ++ "\n"
             where
             unlit End         = []
             unlit (Lit l# ls) = MkChar l# : unlit ls

filterset' res p []     = []
filterset' res p (x:xs) = if (p x) && (notin x res) then
                              x : filterset' (x:res) p xs
                          else
                              filterset' res p xs
                          where
```

```
                        notin x []     = True
                        notin x (r:rs) = nelitpr x r && notin x rs

                        nelitpr (p,q) (r,s) = nelits p r || nelits q s

                        nelits End         End         = False
                        nelits (Lit x# xs) (Lit y# ys) = neChar# x# y#
                                                         || nelits xs ys
                        nelits _           _           = True

insert x# End         = Lit x# End
insert x# (Lit y# ys) = if eqChar# x# y# then Lit y# ys
                        else if ltChar# x# y# then Lit x# (Lit y# ys)
                        else Lit y# (insert x# ys)

nontautclause (cs,as) = nointersect cs as
                        where
                        nointersect End         as = True
                        nointersect (Lit c# cs) as = notmember c# as
                                                     && nointersect cs as
                        notmember c# End          = True
                        notmember c# (Lit a# as) = neChar# c# a#
                                                   && notmember c# as

parse' []      s = redstar s
parse' (' ':t) s = parse' t s
parse' ('(':t) s = parse' t (Lex '(' : s)
parse' (')':t) s = parse' t (x:s')
                   where
                   (x : Lex '(' : s') = redstar s
parse' (c:t)   s = if inRange ('a','z') c then
                      parse' t (Ast (Sym (case c of MkChar c# -> c#)) : s)
                   else if spri s > opri c then parse' (c:t) (red s)
                   else parse' t (Lex c : s)

unicl = filterset nontautclause . map clause
```