Fresh Techniques for Memory Profiling of Lazy Functional Programs

Firas Faisal Moalla
MSc by Research
University of York
Computer Science

September 2015

Abstract

Lazy functional languages are known for their semantic elegance. They liberate programmers from many difficult responsibilities, such as the operational details of computations including memory management. However, the productivity and elegant semantics provided by lazy functional languages do not come without a cost. *Lazy functional programs often suffer from unpredictable space leaks*. For over two decades, various lazy functional implementations have been equipped with memory profiling tools. These tools furnish programmers with valuable information about space demands, but there is still scope for their future development. This dissertation presents two variants of memory profiling tools. The first tool is a *hotspot* heap profiler which presents information in two forms: profile charts and highlighted *hotspots* by source occurrence. The profile chart represents a *hotspot-construction* profile, distributed by hotspot *temperatures*. Hotspots are also marked in the textual display of source programs with the temperature they represent. Further information about hotspots is given in individual profiles. The second tool is a stack profiler which yields information about producers and construction of stack frames.

Contents

Abstract **2**

Contents **3**

List of Figures **6**

Acknowledgements **8**

Declaration **9**

1 Introduction **10**
- 1.1 Motivation . . . 10
- 1.2 Goals and Contributions . . . 11
- 1.3 Outline of Chapters . . . 11

2 Review: Memory Profiling for Lazy Functional Programs **13**
- 2.1 Memory Profiling . . . 13
- 2.2 Lazy Functional Languages . . . 14
  - 2.2.1 Aspects of lazy functional languages . . . 14
  - 2.2.2 Current state-of-the-art space performance . . . 16
  - 2.2.3 Causes of space leaks . . . 17
  - 2.2.4 Profiling complications . . . 17
  - 2.2.5 Early memory profiling implementations . . . 19
- 2.3 Heap Profiling for Lazy Functional Programs . . . 20
  - 2.3.1 Producer and construction profiles . . . 21
  - 2.3.2 Retainer and lifetime profiles . . . 22
  - 2.3.3 Biographical profile . . . 23
  - 2.3.4 Combining heap profiles . . . 24
  - 2.3.5 Cost-centre profiling . . . 27
- 2.4 GHC Profiling . . . 29
  - 2.4.1 Current heap profiles examples in GHC . . . 30
  - 2.4.2 Relation to space leaks . . . 33
- 2.5 Discussion . . . 34

3 *Pure*: A Lazy Functional Language Implementation **35**
- 3.1 *Pure* Implementation . . . 35
- 3.2 The *Core* Language . . . 36
- 3.3 The *Core* G-Machine . . . 37

4 A Hotspot Profiler **40**

4.1 Overview 40
4.1.1 Occurrences 40
4.1.2 Design outline 41
4.1.3 Profiling technique 42
4.2 Implementation 46
4.2.1 Overview 47
4.2.2 Compiler instrumentation 47
4.2.3 Runtime instrumentation 50
4.2.4 The ME2G program
— Post-processing & graphical display 53
4.3 Results 54
4.3.1 The `maxc` program 54
4.3.2 The `execute` program 64
4.3.3 Evaluation 69

5 A Stack Profiler 71
5.1 Motivation 71
5.2 Stack Profiling 71
5.3 Tagging the Stack 73
5.4 Implementation 75
5.5 Results 77
5.5.1 The `sumChops` program 77
5.5.2 The `queens` program 81
5.5.3 Evaluation 89

6 Clausify Revisited 92
6.1 Motivation 92
6.2 The `clausify` Program 92
6.3 Comparing `clausify` Memory Profiles 94
6.4 Evaluation 102

7 Conclusion 103
7.1 Results Summary 103
7.2 Future Work 104

Appendices 106

A The G-Machine 107
A.1 Compilation Rules 107
A.1.1 Scheme $\mathcal{F}$ (Function Definition) 107
A.1.2 Scheme $\mathcal{R}$ (Return Value) 107
A.1.3 Scheme $\mathcal{E}$ (Evaluate) 108
A.1.4 Scheme $\mathcal{C}$ (Construct Graph) 108
A.1.5 Miscellaneous schemes for `case` expressions 109
A.2 State Transition Rules 109
A.2.1 Heap and stack operations 110
A.2.2 Evaluation instructions 110
A.2.3 `case` and `let(rec)` instructions 111
A.2.4 Arithmetic and comparison instructions 112

A.2.5 Printing instruction 112
A.2.6 Initial and final G-machine state 113

B ME2G **114**

C The `clausify` Program **115**

Bibliography **118**

List of Figures

2.1 Lines from a memory cell represent a different cell property. 25
2.2 An ideal heap profiler; lines between heap profiles depict restrictions. 25
2.3 The `sumsList` program. 30
2.4 A producer profile of `sumsList`. 31
2.5 A construction profile of `sumsList`. 32
2.6 A retainer profile of `sumsList`. 33
2.7 A biographical profile of `sumsList`. 33

3.1 Pure implementation phases. 35
3.2 Syntax of the *Core* language. 37
3.3 G-machine reduction of `(square 5)`. 38

4.1 The definition `maxcc` where `Cons` occurrences are annotated. 40
4.2 The definition `maxcc` where every occurrence is annotated. 41
4.3 A hotspot profile example. 44
4.4 Lifetime profile of an individual hotspot. 45
4.5 Modified compiler for occurrence profiling. 47
4.6 Annotation construct in *Core* (similarly for *Pure*). 47
4.7 Occurrences annotation example in an abstract syntax tree. 48
4.8 The $\mathcal{E}$ scheme instrumented for occurrence tags. 49
4.9 The $\mathcal{C}$ scheme instrumented for occurrence tags. 49
4.10 Heap cells annotated with occurrence tags. 50
4.11 Tagged heap cell. 51
4.12 A structural prototype of the `ME2G` program. 53
4.13 The `maxc` program. 55
4.14 A producer profile for `maxc`. 56
4.15 A construction profile restricted by the producers `maxcc` and `take`. . 56
4.16 A hotspot profile for `maxc`. 57
4.17 An individual profile for the hotspot `maxcc`. 58
4.18 A hotspot profile for `maxc` after tail-call optimisation. 59
4.19 A hotspot profile for `maxc` after introducing `listOf`. 60
4.20 A hotspot profile for `maxc` after an occurrences counter `oc` has been introduced. 61
4.21 An individual hotspot profile for `+` after introduction of an occurrences counter `oc`. 62
4.22 The final hotspot profile of `maxc` after introducing `seq`. 63
4.23 The `execute` program. 65
4.24 A hotspot profile of `execute` program. 66
4.25 An individual hotspot profile of the `pop` hotspot. 67
4.26 A hotspot profile of `execute` program after strictifying the stack. . . 68

4.27 A hotspot profile of `execute` program after removing the `code` argument from `value`. 68
4.28 A hotspot-construction profile for the `Taut` program. 70
4.29 A construction profile for the `Taut` program. 70

5.1 A stack profile example. 73
5.2 The stack configuration of an executing function. Some components of the stack frame are excluded to simplify discussion. 74
5.3 An example of a tagged stack frame. 75
5.4 The `sumChops` program. The function `chop` is based on the function `chop8` in [2, Section 7.6]. 77
5.5 A construction profile for `sumChops`. 78
5.6 A producer profile for `sumChops`. 79
5.7 A construction profile after introducing the function `foldl`. 80
5.8 A producer profile after introducing the function `foldl`. 80
5.9 A construction profile after strictifying the application of `sum`. . . . 81
5.10 The `queens` program. 82
5.11 A heap construction profile of `queens`. 83
5.12 A heap producer profile restricted to `Cons`. 83
5.13 A stack construction profile of `queens`. 84
5.14 A stack producer profile restricted to `length`. 84
5.15 A stack producer profile after introducing `length'`. 86
5.16 A heap producer profile after introducing `length'`. 86
5.17 A stack producer profile after modifying `safe'`. 88
5.18 A stack producer profile after removing `append`. 88
5.19 A heap producer profile after removing `append`. 89
5.20 A stack construction profile of `compile`. 90
5.21 A heap construction profile of `compile`. 91
5.22 A construction profile produced using the GHC (version 7.4) profiling system with stack space included. This profile is of the `queens` program with 10 queens as input. 91

6.1 A heap producer profile of `clausify`. 95
6.2 A heap construction profile of `clausify`. 95
6.3 A stack construction profile of `clausify`. 96
6.4 A stack producer profile of `clausify`. 96
6.5 A heap construction profile after introducing `filterset`. This profile corresponds to Figure 11 of [1]. 98
6.6 A heap producer profile restricted to the producers of `Dis`. This profile corresponds to Figure 12 of [1]. 98
6.7 A hotspot profile of `clausify` after introducing `filterset`. 100
6.8 A hotspot profile after improving `disin`. 101

Acknowledgements

First, I thank my parents for their continuous support and help. I would also like to thank my supervisor, Prof. Colin Runciman, for his guidance, help and support in my research. My sincere thanks also goes to Jose Calderon who provided help and answered my questions. Thanks to Defaf for her help in drawing figures.

Declaration

Chapter 1

Introduction

1.1 Motivation

Lazy functional languages are known for their semantic elegance. They support an expressive and modular style of programming. They liberate programmers from many difficult responsibilities such as the operational details of computations and — of particular relevance for this dissertation — memory management. Many features of lazy functional languages such as high-level abstractions and expressiveness can increase the productivity of programmers.

However, the productivity and elegant semantics provided by lazy functional languages do not come without a cost. *Lazy functional programs often suffer from unpredictable space leaks*. The high-level abstraction, together with the demand-driven execution to realise so-called lazy evaluation and the automatic storage management, make it difficult to reason about the space performance of programs.

Fortunately, for over two decades, lazy functional implementations have been equipped with *heap profiling* tools. Heap profiles furnish programmers with valuable information about space demands. Observations recorded in heap profiles often help programmers to change just a few parts of a program in a way that reduces the space consumption dramatically. However, even with such tools it can be difficult to reason about the memory performance of programs. By default, profiling results report sources of demand for memory by referencing units such as top-level functions. This provides concise summaries of memory since expression-level details are abstracted away from profile results. However, there is a consequent gap between expression-level reasoning and profiling results.

Although memory resources have been increasing in modern computers, 21st-century programmers do still complain about the space consumption and difficulty of reasoning about memory behaviour, of even small programs. For example:

> *"When programming in Haskell (and especially when solving Project Euler problems, where suboptimal solutions tend to stress the CPU or memory needs) I'm often puzzled why the program behaves the way it is. I look at profiles, try to introduce some strictness, chose another data structure, ... but mostly it's groping in the dark, because I lack a good intuition."* [3]

Also, efficient use of memory is still a crucial concern in some real-world applications. For example, some protocols that are used in e-commerce restrict memory resources to a limit (e.g. 8 kb) [4].

This dissertation develops variants of memory profiling tools. The motivation is that profiling functional programs at the expression-level may help programmers reason about the space behaviour of programs better. However, functional programs are composed of many expressions. If every expression is profiled, the programmer could be overwhelmed by the amount of profiling information. The goal of this dissertation is to show that this does not have to be the case. We develop heap profiling techniques which approximate the relation between heap profiling results and expressions. We suggest a profiling technique which locates dominant sources of demand for memory in the form of specific constructions of function applications within expressions.

What more might be done to improve memory profiling? Let us not forget the stack. Stack overflows are common in lazy functional languages. Programmers often complain about excessive stack demands. So there is good reason to develop a stack profiler which yields information similar to heap profiles.

1.2 Goals and Contributions

Our overall goal is to advance profiling tools for analysing memory use in lazy functional programs. More specifically, the purpose of these tools is to enable programmers to understand better the space behaviour of programs.

The specific contributions of this dissertation are:

- ***Hotspot profiler*** We present a *hotspot* heap profiler which provides information about the source components of a program that are associated with high memory demands (termed *hotspots*). Information about hotspots is presented both in profile charts and in annotated source programs. The profile chart shows memory information about a hotspot. Hotspots are also annotated in textual displays. Further information about hotspots is given in individual profiles. In this way, a hotspot profile relates heap profiling results to fine-grained symbolic constructions of programs.

- ***Stack profiler*** As an auxiliary contribution, we present a profiler which shows similar charts for stack memory and the associated program components.

1.3 Outline of Chapters

Chapter 2 introduces some general types of memory profiling tools. We then discuss modern lazy functional languages and why reasoning about space performance is difficult in such languages, along with causes of space leaks, current space performance of an optimising compiler and why profiling lazy programs is difficult. We then review heap profiles in lazy functional implementations, finally explaining the current heap profiles provided in GHC, a state-of-the-art implementation of Haskell. We conclude this chapter by summarising problems of memory profiling in lazy functional implementations, setting out proposals in this context.

Chapter 3 presents the functional language implementation for which we implement memory profiling tools. A high-level definition of an intermediate language is given. We then describe an abstract machine for the functional language. (A more formal specification of the abstract machine is provided in Appendix A for this dissertation to be self contained).

Chapter 4 introduces a hotspot profiling tool which annotates hotspots of memory demand occurring in programs. We explain the nature of hotspot occurrences and why we profile programs in this way. We set out goals and decisions which led to a hotspot profile and give an outline of the hotspot profile design. We describe the technique to collect profiling data for its graphical presentation with the reasons behind them. We further describe our implementation of a hotspot profiler along with details of modifications to our functional implementation to support hotspot profiling. We give brief account of the implementation of hotspot graphical presentation. Finally, we show how iteratively applying hotspot profiling can guide program modifications to reduce memory demands, illustrating both some advantages of hotspot profiling and an example of its drawbacks.

Chapter 5 starts by outlining the motivation behind stack profiling. We briefly discuss how the stack is used in our implementation and introduce the profiling technique and components of the stack profiler. We then describe a high-level scheme to attach information to the stack for profiling and outline our implementation of the stack profiler. We briefly show illustrative results of applying stack profiling and discuss the results.

Chapter 6 revisits the `clausify` program. It first sets out our motivation for profiling `clausify`. We explain this program in summary. We then compare the original heap profiles with our stack and hotspot profiles; in particular, we show how stack and hotspot profiles yield further information.

Chapter 7 concludes and discusses the drawbacks and future work on hotspots and stack profiling.

Chapter 2

Review: Memory Profiling for Lazy Functional Programs

In this chapter we review memory profiling techniques for lazy functional programs. This chapter is organised as follows. Section 2.1 briefly describes general types of memory profiling tools. Section 2.2 discusses aspects of lazy functional languages including performance and profiling difficulties. Section 2.3 reviews several heap profiling techniques of lazy functional languages. Section 2.4 demonstrates the current state of heap profiles. Section 2.5 concludes with a discussion on heap profiling. Throughout, we assume that the reader is familiar with functional programming. For those without such a background, the book in [5] is an excellent tutorial.

2.1 Memory Profiling

Programmers wishing to improve the space efficiency of software normally use profiling tools to identify the program components that are responsible for space faults. Cheap replacements of such components can increase the program's performance significantly. This cyclic refinement strategy is common in software development [6].

The sort of memory profiler resorted to depends on the type of the space fault, which in turn depends on the nature of the programming language. The following is a general classification of the memory profiling tools which are commonly used to identify sources of space faults [7][8]:

- *Allocation profiles* are the most primitive type of memory profiling. They record the total amount of dynamically allocated memory during the execution of a program. Memory allocation counts are attributed to components at the source level (e.g. functions/procedures). Usually, this information is presented in a textual table. Sophisticated tools of this kind report further information such as the distribution of memory allocation between functions through a call graph. Using this information, one can reduce allocation counts. However, this may not improve the space performance; a function causing significant allocations is not necessarily inefficient, as allocated objects may be reclaimed thereafter. Likewise, seemingly insignificant allocation counts might be the source of long-lived memory objects. An allocation count can be a deceptive representation of memory demands.

- *Memory leak profiles* are more specific to programming languages with explicit dynamic storage management (e.g. C). These profilers are specialised for detecting leaked memory in the dynamic store. Among other storage parameters, profiling information includes the amount of leaked memory along with a call trace showing the components responsible for allocating objects which are never freed.

- *Heap profiles* are common for programming languages with automatic memory management. Heap profiling tools report information about the content of the *live* heap during program execution. Generally, the profiling information is attributed to the source program and represented to the user in a graphical summary. Heap profiling is a more general instrument, giving several different views of live objects in memory. At one extreme, profiling results include primitive information such as the program component responsible for allocating live heap objects or their eventual lifetime. At the other extreme, heap profiles can reveal space leaks through sophisticated views of memory. An example is memory objects which are unnecessarily retained in the heap. With such information, one can detect the source of space leaks and reduce the live memory demands.

The context of this dissertation is memory profiling and space performance of lazy functional programs.

2.2 Lazy Functional Languages

This section reviews several aspects of lazy functional languages with emphasis on space consumption. Section 2.2.1 briefly describes some of the benefits and drawbacks of functional languages. Section 2.2.2 reviews the current state of space-performance. Section 2.2.3 discusses some of the causes of space leaks. Section 2.2.4 outlines some of the difficulties of profiling lazy programs. Section 2.2.5 concludes with early implementations of heap profilers.

2.2.1 Aspects of lazy functional languages

Lazy functional languages provide a powerful method of computation, liberating programmers from many tedious responsibilities. They lead to a style of programming where concise, expressive and modular programs are often written with ease.

Here follow some of the features of *modern* lazy functional languages which contribute to the *modularity*, *expressiveness* and *correctness* of programs.

- **Declarative programming**. Lazy functional languages belong to the *declarative programming* paradigm. They liberate programmers from the underlying computer architecture through high-level abstractions. Programmers can concentrate on expressing solutions to problems in a declarative style, without being concerned about low-level operational details. Some abstraction helps programmers to write short and expressive programs. The elegance of declarative programming in relation to lazy functional programming is demonstrated in Nilsson's thesis [9].

- **Higher-order functions**. Functions which either take or return functions are *higher-order*. This idea is typically implemented by promoting functions as *first-class citizens*. Higher-order functions can be generalised over a structured type and specialised with many base functions. This concept provides a powerful method of modularity, as illustrated by Hughes in [10].

- **Lazy evaluation**. A strategy which evaluates arguments to functions only when their value is actually required, and then at most once, is called *lazy evaluation* (often named *call-by-need*). Lazy evaluation adds a new power of expressiveness to the language allowing, for example, the creation of *infinite data structures* [11]. Furthermore, programmers can rely on the underlying implementation to prevent the price of unneeded and repeated computations [12]. Lazy evaluation is the default evaluation scheme in non-strict functional languages.

- **Pure computations**. Pure functional languages lack side-effects; they prohibit *destructive assignments* (or updates). As a consequence, replacing expressions with their values does not change the semantics of the programs. The notion of "equals can be substituted for equals" (termed *referential transparency*) makes programs easier to reason about and maintain [12]. Furthermore, transparency encourages a less error-prone programming style [13].

- **Sophisticated type system**. Functional languages often have strong type systems. Programmers may use types to specify the problems to be solved before implementing solutions [14]. In addition, type checking detects many mistakes early in the compilation process [9].

- **Garbage collection**. Manual storage reclamation increases work for programmers. It is estimated that 40% of the time spent in the development cycle of programs is wasted in resolving storage management issues [15]. Automatic storage reclamation techniques imposed by garbage collectors free programmers from such tiresome responsibilities. In general, garbage collection is an essential component of high-level programming languages.

The many beneficial features of lazy languages can increase the productivity of the software development cycle by several factors compared to their conventional imperative counterparts [7]. Indeed, it is argued that most programmers who have been involved in writing considerable numbers of programs in both imperative and functional styles find the latter more attractive [12].

However, the enhanced productivity and semantic elegance provided by lazy functional languages come with a cost. There are in principle two major drawbacks:

- *Unpredictable execution behaviour*. Although in lazy functional languages it is fairly straightforward to reason about the meaning of programs, it can be difficult to reason about their execution behaviour. The demand-driven execution imposed by lazy evaluation makes the execution order of programs unpredictable. The complex evaluation order, together with the high level language constructs, makes it extremely difficult (if not impossible) for programmers to predict the execution order of large programs. A consequence of this unpredictable evaluation order is difficulty reasoning about space and time demands [6][11]. Implicit allocation and reclamation further complicates the estimation of space demands [16][17] and it is therefore common for lazy programs to suffer from unexpectedly poor performance [11]. Another consequence of the unpredictable execution behaviour is the difficulty of detecting program errors. Being unaware of the execution behaviour can make the process of detecting bugs (e.g. logical errors) troublesome. This problem has been recognised and studied extensively over the past decades (though this topic is not covered in this dissertation [9][18][19][20][21][22][23]).

- *Execution overheads*: The high level of abstraction provided by lazy functional languages introduces extra execution overhead in terms of time and space, particularly the latter. Functional programs often need a great amount of storage to run, mainly due to lazy evaluation; suspending computations to the point where they are actually needed together with sharing may save time. However, this policy can readily produce space leaks: the accumulation and retention of computations in memory in ways which are hidden from the programmer [11][24].

As a consequence of the unpredictable execution behaviour and the execution overheads, *a major weakness of lazy functional programs is that they often suffer from unexpected space leaks* [11]. Writing space-efficient programs in lazy languages is difficult; there is a trade-off between productivity and performance.

2.2.2 Current state-of-the-art space performance

In recent years, much research has been devoted to efficient implementation and optimisation techniques for lazy functional languages. This has led to significant improvements in the performance of lazy functional programs. Notably, lazy functional programs have seen dramatic improvements in terms of execution *speed*, with impressive results. Programs written in Haskell, for instance, can compete with the execution speed of programs written in the equivalent imperative counterparts (e.g. C) [25]. Recent studies show that with clever optimisation techniques such as *generalised stream fusion*, the execution speed of lazy functional programs can beat programs written in low-level languages [26].

Most research efforts have been aimed towards improving speed. Compiling techniques addressing space performance are few compared to those improving speed. The result is space-hungry implementations favouring speed at the cost of space [16]. Worse, speed improvement techniques (e.g. *full laziness transformation* [27]) can introduce space leaks [28]. On the other hand, a few techniques addressing speed can dramatically reduce space consumption; some examples are *deforestation* [29] and *strictness analysis* [30].

Speed may be the first performance concern for software developers. However, for functional programming systems, space use is usually more critical: space leaks in functional programs are notorious [2, Chapter 12][5, Chapter 6][11, Chapter 23][31][32, Chapter 25][33][34, Chapter 7]. Functional programmers often complain about unexpected space leaks. This concern is increasing as lazy functional languages are becoming more common in industry. Sampson in [35] report their experience of writing a commercial application in Haskell using GHC, a state-of-the-art optimising compiler for Haskell. In their concluding remarks they mention that performance was never a problem, *except for space leaks*.

From a practical perspective, space-usage issues have mostly been addressed by the provision of heap profiling facilities [36]. Heap profiling tools reveal space leaks that are otherwise challenging (or even impossible) to locate. With the guidance of such tools, one can improve the space performance of programs. However, even with such a guidance, programmers often find it difficult to reason about space performance, pressing for more sophisticated profiling tools [37][38].

2.2.3 Causes of space leaks

There are many causes of space leaks in lazy functional programs. Conceptually, they can be generalised as follows [36][39]:

- *Degree of evaluation*. Lazy evaluation can save the amount of memory-usage by taking into account only the demanded computations. However without careful use, it can cause an unnecessary accumulation of suspended computations in memory. Sometimes accumulated computations can be avoided by forcing applications at a certain point. However, writing function definitions in an unnecessarily strict manner can cause space faults.

- *Degree of sharing*. Sharing can also increase or decrease memory consumption. Sharing a large structure between expressions may save speed by computing the structure once and saving the results for later re-use. However, sharing a structure can retain computations in the heap due to the shared references. Unsharing may cause the structure to be re-evaluated but this can allow the structure to be computed in constant space.

- *Algorithmic design*. The question of efficiency is highly related to algorithmic choices; these in turn are related to the two previous points. From a space-efficiency perspective, an efficient algorithm is one which avoids constructing long-lived structures in the heap or which reduces the amount of memory for such structures.

Space leaks are not necessarily introduced by the programmer. On some occasions they are the fault of the compiler, for example, by unnecessarily retaining cells in heap memory [40]. Of course, the programmer in this case is not expected to solve the space leak since this requires accessing and understanding the implementation.

2.2.4 Profiling complications

In practice, improving the space efficiency of programs often requires the aid of profiling tools. Generally, there are two critical problems when profiling programs. First, the profiler should pinpoint "hotspots" in programs — by explicit or implicit means. Second, to be meaningful for the programmer, the reported information must be mapped from the code-level back to the source-level. This is fairly straightforward to achieve in conventional imperative languages (except for optimised programs) [41]. In contrast, it is difficult to accomplish for functional languages, especially in the presence of laziness.

From a profiling perspective, the many beneficial features advocated by lazy functional languages are problematic. The following briefly outlines several problems which pose difficulties when profiling lazy functional languages [6][7][8]:

- In functional programming, the leading role of programmers is to construct functions to solve problems. Thus, in principle, a program consists of function definitions [5]. This style of programming usually leads to the production of *many functions*. Presenting information about a large number of profiled functions may be hard to interpret. Rather, it is more useful to provide summarised profiling information which requires aggregation techniques.

- *Polymorphism* invites the programmer to re-use functions in different contexts [42]. This makes it difficult to identify the cost of different application instances. For example, the expression

  ```
  filter predicate list
  ```

  may be used in several places, causing a large proportion of heap consumption. Attributing the costs to `filter` may not reveal the expensive instance(s) of `filter`. Techniques which distinguish different instances of applications may be essential. One possible solution to this problem is to attribute the cost of `filter` calls to its application sites.

- *Higher-order functions* provide an abstract mechanism to encapsulate common patterns as functions [2][13]. From a profiling point of view, the higher-order argument being applied may be difficult to determine at compile time. This problem is tackled in [43] by introducing the notion of *current function*: each function is associated with an auxiliary variable so when a function is entered (i.e. starts its execution), an interrupt handler charges the execution costs to the auxiliary variable of the current function being executed. Subsequently, costs are fairly distributed to every function, including higher-order functions. Consider the following equation, for example.

  ```
  f bop xs = filter (10 bop) xs
  ```

  Here, the costs of executing the higher-order argument `bop` are attributed to `bop`. However, the cost of an unprofiled function is charged to its caller. So if `bop` is an unprofiled function then its execution costs are charged to `filter`. The idea of current function was originally used for a strict functional language (ML) though it is adopted for profiling lazy functional languages.

- *Lazy evaluation* ensures that expressions are *only* evaluated when their value is required. The execution flow of programs is thus driven by the context in which expressions are demanded. Accordingly, the evaluation order of expressions may not correspond to the way in which programs are written. This complicates the task of attributing costs to the relevant components of programs. For further illustration, consider the following equations.

```
d x   = x * x;
f x y = d x + y;
g     = f expr1 expr2;
```

Under a *strict* evaluation strategy, the costs of evaluating $expr_1$ and $expr_2$ might reasonably be attributed to the top-level function g since both expressions are completely evaluated before the function f is applied. This poses no profiling problems since the reported execution costs correspond to the syntactic unit in which $expr_1$ and $expr_2$ are evaluated. In a *lazy* evaluation strategy, however, $expr_1$ is demanded in d (instead of g). Similarly, $expr_2$ is demanded in f. Should the costs of $expr_1$ and $expr_2$ be attributed to g or to d and f, respectively? The former attribution scheme is termed *lexical scoping* while the latter strategy is known as *evaluation scoping*. It is argued that lexical scoping is more practical than evaluation scoping for lazy functional languages [7][44].

The problem of profiling lazy languages is further complicated by the sharing policy. Several expressions may require the value of a shared expression; the first expression causing the evaluation is virtually in charge of the costs. Subsequent expressions profit from the results for free. This policy raises difficulties for distributing costs between functions that share an expression, especially in the presence of *constant applicative forms* since they are usually treated as "global" functions. Breaking down the costs between functions that demand the results in a fair manner requires sophisticated analysis.

- Typically, compiling lazy functional programs includes successive *transformations* and *optimisations* [45]. At an early stage, the compiler performs radical transformations on the original source code. The transformations are necessary to support the provided high-level abstractions and to produce efficient code. The resulting object code bears no obvious relation to the original source code. Relating the low-level code back to the original program is a major task: the structure of the original program must be maintained throughout the transformation phases, taking into account the implicit auxiliary functions brought by the program transformation phases.

As a result of research to address the above profiling difficulties, implementations of the lazy functional language Haskell, for instance, are equipped with specialised profiling tools (see Section 2.3).

2.2.5 Early memory profiling implementations

Real memory profiling tools[1] were originally developed by programming language implementers to analyse the characteristics of dynamic memory management techniques. Measurements gathered from memory profiles were studied in order to improve implementation techniques of the storage manager or to gain insight about the nature of programming language features. A few examples of storage profiling systems which follow this sort of study are briefly discussed below.

[1]We disregard memory *allocation* count profilers since they fail to characterise live memory demands.

Ripley, Griswold and Hanson developed a heap profiling system for SITBOL, an interpreted implementation of SNOBOL4 [46]. Their heap profiling system measures information about allocated memory blocks; each memory block is composed of a memory object and a heading section. The header contains *creation histories* which include the size, type (e.g. `string`) and the *lifetime* of the memory block in addition to the source line number of the operation causing the allocation of the memory block. Creation histories are recorded at every garbage collection and *post-processed* to generate a summary profile in table form. The profile table is divided using block types, followed by several storage parameters including space consumption and the *average block lifetime*. Profiling results in their experiment were used to improve application programs and storage management techniques manually. In their conclusion, Ripley *et al.* advocate the value of profound memory analyses. However, they claim that storage performance measurements are more useful for improving memory implementation techniques than for optimising application programs.

Hartel and Veen instrumented an interpreter for a lazy functional language called SASL to analyse several properties of *combinator graphs* in the course of reduction [47]. Their approach involves gathering statistics about graphs at regular intervals in order to observe the performance of graph reduction under certain programming language features (e.g. *sharing*). The aim of their experiment is to determine implementation techniques that are suitable for a special purpose parallel machine, relying on measurements gathered from different programs ranging in size and complexity. Statistics involve the size of graphs, distribution of sharing and the lifespan of nodes. Information is collected at program-level (excluding sub-components of programs). From this information they observed that most nodes are shortly lived, which assures the suitability of generational garbage collectors for lazy functional languages. A very similar study on a lazy functional compiler is that of Wild, Glaser and Hartel [48].

A much more recent study is by Hertz, Blackburn, Moss, McKinley and Stefanović [49]. They present algorithmic methods for computing the lifetime of objects as an alternative to *granulated* traces to simulate the performance of garbage collection in object-oriented languages. The ultimate goal of their experiment is to produce accurate measurements of objects' lifetime in order to optimise garbage collection methods.

In principle, these studies share a similar trend: to investigate the characteristics of programming language features and the performance of storage management techniques with the goal of improving implementation methods. Furthermore, they stress the value of memory statistics for storage performance analysis. However, their memory profiling tools are designed for *implementers* wishing to study compiling architectural options and not for programmers hoping to examine the space efficiency of their programs [36].

2.3 Heap Profiling for Lazy Functional Programs

This section reviews heap profiling techniques of lazy functional languages. Section 2.3.1 describes producer and construction profiles. Section 2.3.2 describes retainer and lifetime profiles. Section 2.3.3 discusses biographical profiles. Section 2.3.4 describes some combinations of heap profiles. Section 2.3.5 explains cost-centre profiling techniques.

2.3.1 Producer and construction profiles

Runciman and Wakeling designed the first heap profiler for use by *programmers* [1]. Their heap profiling system is hosted in an implementation based on LML, a lazy functional language [50]. The aim of their heap profiling system is to provide heap consumption summaries as a guide for both programmers and implementers to observe and improve the space performance of programs. Two kinds of heap profiles are implemented: a *producer profile* identifying the function responsible for producing heap cells and a *construction profile* describing heap cells in terms of the values they represent.

The implementation of this tool involves two components: a modified compiler which produces profiling data at runtime and a post-processor that displays these data as a graphical chart.

Profiling information about producers and constructions is determined at compile-time [51]. Preserving this information during the compilation phases poses profiling difficulties since the compiled program radically departs from its original representation. For example, at the *lambda lifting* stage, some parts of the functions body are turned into *supercombinators* with a compiler-generated function name. Since these names can be meaningless to the programmer, the modified compiler manages to include the original function name in the compiler-generated name. Thereafter, the profiler restores the function names to their original form.

At the runtime level, every heap cell is enlarged with an extra field to accommodate *static cell tags*; each tag is a pointer to information determined at compile-time. This information includes the producer and the construction of the heap cell. Tags are initialised to heap cells on allocation.

When a heap profile is requested, the implementation suspends the execution at specified regular intervals. At each interval, the *live* heap is traversed to derive information from live cells. The gathered information is logged into a file before resuming the execution. After a program completes its execution the information contained in the log file is post-processed by a separate program into a graphical chart (also termed a *heap profile*).

A heap profile represents the variation of the occupied heap space (in bytes) over the time the program takes to execute (in seconds). Heap consumption is depicted by shaded bands which are associated with keys. Each key corresponds to a program component in the original source code. Careful attention is paid to the representation of heap profiles. For example, keys occupying less than 1% of the total space are omitted from the graph to focus on bands occupying the most space.

To concentrate on a specific heap *section*, heap profiles can be restricted through profiling options. An example is a construction restricted by a specific producer.

Runciman and Wakeling applied their heap profiling tool to a small program (`clausify`) consisting of 130 lines of code. Repeatedly applying the producer and construction profiles to the target program together with profiling restrictions aided in discovering several space faults at both the source program and compiler levels. The final result was a reduction on memory demands from a peak of 1.3 Mb to 7 kb.

However, it was soon realised that heap profiling by individual producers and constructions is likely to be impractical for large programs. In response, Runciman and Wakeling [24] extended the heap profiling scheme in [1] to cope with complex programs. In particular, producer profiles were upgraded to support producers by a module or a group of modules. Similarly, construction profiles were upgraded to support constructions by type.

The extended heap profiling scheme was successfully used to reduce the space consumption of an LML compiler comprising 16,000 lines of code (and 200 modules) by a factor of two. Similar applications of heap profiling with satisfying results are reported in [1].

The work by Runciman and Wakeling [1][24] started a fresh line of work in the field of functional languages and memory profiling. Their practical tool brings an insight into the space behaviour of lazy functional languages and guides programmers to improve space costs. Furthermore, the idea of heap profiling for programmers opened a fresh field of research on practical profiling.

However, the profiling scheme in [1][24] suffers from two problems. As noted by the authors, their profiling scheme cannot distinguish between different application instances. For example, there could be many applications of (say) `map` in different locations of the source program, but which instance of `map` corresponds to an identifier in a heap profile? This problem has been criticised in the literature [7][8][52]. Distinguishing producers by instance can be useful for large programs; however, this requires a sophisticated cost attribution scheme (see Section 2.3.5).

2.3.2 Retainer and lifetime profiles

Runciman and Röjemo [17] extended the heap profiling system in [24] with two heap profiles hosted in NHC, a space-efficient Haskell compiler [53].

The first profile is a *retainer profile* which classifies heap cells by the program components which have *direct* access to them [17][36]. Such program components are termed *retainers* and are represented in a *retainer set*.

Program components are counted as retainers if they are one of the following candidates: (1) function closures — residing in the heap or the stack; (2) *constant applicative forms*. When a retainer set for a cell is to be computed, retainers are those that have a path to a cell without passing any retainer candidate encountered in the path. Program components occurring as *weak head normal form* are not treated as retainers. Forbidding constructors from being counted as retainers is effective in practice. For example, knowing that a `Cons` cell is retained by another `Cons` cell might not be helpful to the user. Instead, the retainer of the original `Cons` cell is counted as the retainer of the entire successors of the original `Cons` cell. As an example, consider the following expression.

```
map (++"\n") ["line1","line2"]
```

Here, `map` retains two components: the list `["line1","line2"]` and the closure ++. The list `"\n"` is retained by the closure ++ even though `map` has a path to `"\n"`:

$$\texttt{map} \rightarrow \texttt{++} \rightarrow \texttt{"\textbackslash n"}$$

The reason ++ is counted as the retainer of `"\n"` is that ++ is a retainer candidate which is encountered along the way from `map` to `"\n"`.

To profile retainers, every heap cell is extended with a word to accommodate a retainer set. When a retainer profile is requested, the live heap is traversed at regular intervals and each cell is tagged with its set of retainers.

A heap cell can be retained by more than one program component due to *sharing*. As a consequence, gathering the set of retainers for many functions with *shared references* requires multi-traversal of the heap. This can impose expensive profiling overheads in terms of space and time. Efficient algorithmic methods for computing the set of retainers in both space and time are addressed in [17].

To reduce profiling overheads, the number of heap traversals is limited by a profiling option which restricts the elements of retainer sets to a specific number.

Profiling *restrictions* from a retainer profile to a producer or a construction profile is provided to the user. However, restricting a retainer profile to a lifetime profile (discussed below) is omitted since the retainer of a memory cell can change through its lifetime. This poses problems from a profiling perspective. We discuss this in Section 2.3.4.

The second profile is a *lifetime profile* that characterises heap cells by their eventual *lifetime* in a program. On allocation, each memory cell is labelled with its *creation time*. Creation times are measured by the number of heap censuses that have occurred during profiling. After a program is executed for a lifetime profile, the censuses in the log file contain heap cell counts associated with creation times. The lifetime of heap cells are derived by post-processing creation times in the log file [16][17].

In [1], Runciman and Wakeling have reduced the memory peak to 7 kb for the `clausify` program. Possibilities for further space reduction were not apparent. However, the information provided by retainer profiles allowed Runciman and Röjemo to lower the peak memory from 7 kb to 1 kb [17].

Retainer profiles are useful to discover space faults brought by functions retaining unnecessary data or those due to excessive sharing. This sort of information is not provided by producer and construction profiles [1][24]. In the retainer profiling scheme described here, retainers are not treated uniquely; there is no mechanical way of identifying retainers that *consume* computations.

A lifetime profile can suggest space leaks. For example, a large band of long-lived cells may suggest lazy accumulation of computations or potential dragging. The suspected heap band can then be inspected by different kinds of heap profile. This profile led to the development of a biographical profile (see Section 2.3.3).

2.3.3 Biographical profile

Runciman and Röjemo [39] extended their original heap profiling tool in [17] with a *biographical profile* that provides a simple model to classify memory-cells according to their usefulness in a program. This profile classifies memory cells according to the biography of heap cells. A heap cell is classified as [39]:

- *lag* from the time it is created until its first use.
- *use* from the time of its first use until its last use.
- *drag* from the time it is last used until it is destroyed.
- *void* if it is created but never used during its lifetime.

A heap cell is used if it contributes to a computation which involves function applications, primitive operations or case analysis.

A biographical profile can suggest space leaks. For example, drag and void suggest heap waste since heap cells are never used (again) in the program. Lag suggests delayed computations causing memory leaks which can be reduced by either creating them later or forcing computations. However, use is an over-estimate; the frequencies between the first use and last use are ignored.

The implementation of the biographical profile extends heap cells to host three time-stamps:

1. The creation time of the cell.
2. The first use time of the cell.
3. The most recent use time of the cell.

Identifying the biography of heap cells requires full knowledge of how these cells are used in the future: the first and most recent use of a cell requires future examinations of censuses and *retrospective* extensions to former censuses. Therefore, heap cells are examined *post-mortem*: at every census, information about *dead cells* is logged to a file. This information includes a sequence of cell counts for every unique time-stamp of dead cells; that is, creation time, first use and last use. Information about heap cells which are destroyed between censuses is recorded, contributing to the next census summary. When a program terminates, the biographical data of heap cells are derived by a post-processor. The post-processed information is presented in a graphical form.

Restrictions from biographical profiles to static properties of memory (e.g. the producer or construction of a lag) are supported. However, no combination of biographical and retainer profiles is allowed.

The biographical profile allowed the authors in [39] to detect space leaks in programs which claimed to be space-efficient. For instance, most of the heap usage of NHC was found to be lag, drag and void. Using the biographical profile in combination with producer and construction profiles, the authors reduced the space consumption of NHC by a factor of two.

Unlike other heap profiles (e.g. producer and retainer), biographical profiles can point directly to a space fault. The classification of lag, drag, void and use can reveal space faults that are otherwise difficult to detect. This classification is adopted by other profiling systems for different programming languages (see Section 2.5).

2.3.4 Combining heap profiles

Heap profiles classify heap content by means of memory attributes. Static memory attributes, for instance, show heap memory consumption by producers or constructions. Through memory attributes, one can view different properties of memory cells.

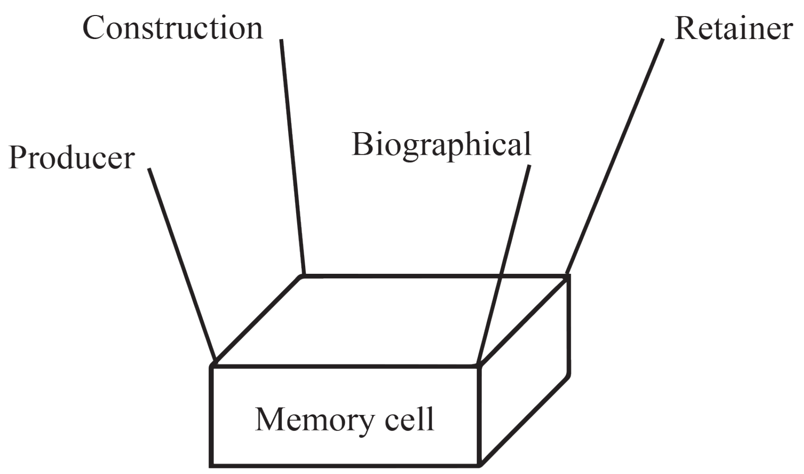


Figure 2.1: Lines from a memory cell represent a different cell property.

But this is not enough as each property is isolated from the others.

Programmers examining a heap profile will likely question a section of the heap to pin-point the reason for memory-demand excesses. If a heap profile shows a large drag band for instance, one may ask several questions: "Who is retaining these dragged cells?", "Who is producing them?", "What value do they represent?" By viewing a section of the heap through a *combination of memory attributes*, one can often diagnose the cause of heap memory waste. Each memory attribute combination corresponds to a profile *restriction*.

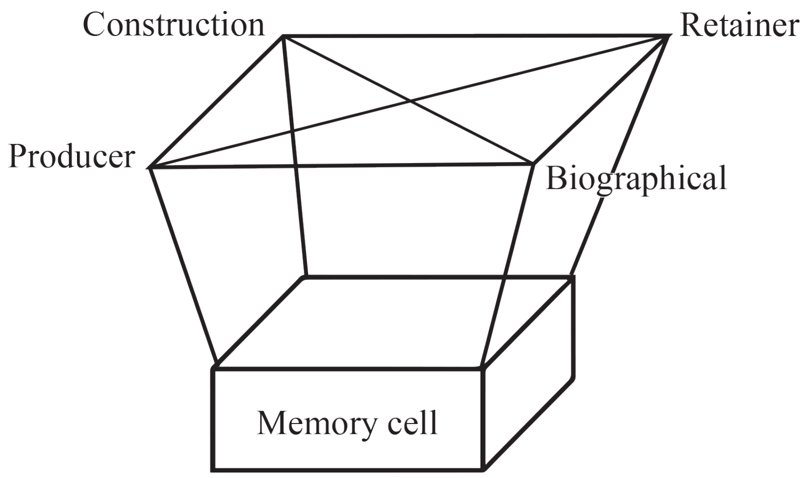


Figure 2.2: An ideal heap profiler; lines between heap profiles depict restrictions.

As Runciman and Röjemo stress [51], an ideal heap profile should therefore allow all kinds of profiling restriction by combining memory attributes. However, this ideal is not straightforward to achieve for all combinations of heap cell attributes. Below is a summary, classifying memory attributes by distinct heap profile [51]:

- *Producer and construction profiles*. These are static attributes of memory; they are identified at compile-time and stored permanently in heap cells — *invariant* attributes.

- *Retainer profile*. Retainers are determined at runtime, which makes them dynamic attributes of memory cells. However, these dynamic attributes are not invariant: the retainers of a cell may change due to the mutating heap structure. Furthermore, a cell retainer can be identified by inspecting the *live heap* at one point; it is an *instantaneous* class of memory attribute.

- *Biographical profile*. Memory attributes identifying biographical information are dynamic and *post-mortem*: information is gathered from the *dead heap* through post-mortem censuses. Biographical attributes are neither invariant nor instantaneous: determining the biographical class of cells requires future and retrospective examinations of censuses.

Combining biographical or retainer attributes with producer or construction attributes is straightforward since the latter are static-invariant attributes of memory. So even with post-mortem techniques, these static restrictions pose no difficulties. However, a combination of post-mortem attributes with instantaneous ones which are based on live heap techniques is problematic.

Prohibiting a restriction between biographical profiles and retainer profiles can be inconvenient for the user. Consider a group of producers X obtained by restricting a producer profile to the drag classification. If X produces cells other than those being dragged, then a retainer profile restricted to the heap section produced by X will not directly point to the retainer component preventing the dragged cells from being reclaimed. The user is left to guess which retainer is the offending one. The obvious solution is to combine biographical and retainer attributes to produce the desired restricted-heap profile but this requires joining live-heap census and post-mortem census profiling.

Retainers are identified by tagging live heap cells with retainer sets, while deriving biographical phases requires tagging dead heap cells with their creation time, first use and last use. This information can be combined by executing a program twice. The first time, a post-mortem examination of censuses is taken to store the times of last use. With the same program input, the program is executed again, but this time, a live-heap census is taken to record information about the retainer set, creation time and the first use (using a use-marker bit). The data recorded from post-mortem censuses compensate for the missing data gathered in the second run.

Care must be taken when the information recorded from the post-mortem pass is used in the second pass; cells which are recorded for the last use information in the first execution must be identified as the *same* cells confronted in the second run. A possible solution as described in [51] involves distinguishing identical cells by numbering live-heap cells while timing censuses intervals by memory allocation counts to ensure that cells are recorded at the exact same points during execution.

Similar techniques can be used to restrict problematic combinations of memory attributes (e.g. retainer to lifetime profiles). Unrestricted combinations of heap cell attributes are necessary for an ideal heap profiler, even at the cost of extra profiling overheads. Indeed, the profiling system in NHC supports *all* kinds of profiling restriction.

2.3.5 Cost-centre profiling

Sansom and Peyton Jones [7][55][56][57] developed a profiler capable of measuring time, space and live heap usage for GHC, an optimised compiler for the Haskell programming language. Although heap profiling is supported, the emphasis is on *time processing*.

In their profiling scheme, expressions are annotated either implicitly (by the compiler) or explicitly (by the programmer) with *cost centres*. A cost centre is a label to which the costs of executing an annotated expression is attributed. The syntax of Haskell is extended with an `scc` (set cost centre) construct to make the association of expressions with a cost centre explicit:

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

From a semantic perspective, the `scc` expression returns the value of *expr*. From an operational view, however, an `scc` expression causes the cost of *expr* to be attributed to the cost centre *label*. Take the following annotated expression as an example.

```
fun y xs = scc "funMap" map (f y) xs
```

Here, the costs of evaluating the expression (`map (f y) xs`) are attributed to the cost centre `"funMap"`. The scope of an `scc` construct extends all the way to the right; however, this scope can be restricted by surrounding the `scc` expression with parentheses. Furthermore, costs attribution are governed by a set of rules. Below, several aspects of the cost centre profiling scheme are summarised informally [7]:

- Since the evaluation demand of expressions depends on the surrounding context, the profiler attributes the costs of evaluating expressions with respect to their *actual evaluation*: the evaluation degree of an expression.

- The evaluation costs of all `scc` expression instances with an identical label are attributed to one cost centre. The instances count is reported together with the total cost of all the instances. For example, the costs of evaluating the instances of (`scc "map" (f y) xs`) are attributed to the cost centre `"map"`.

- The cost of evaluating an expression excludes the costs of evaluating its free variables. For example, in

  ```
  f xs = (scc "last" last xs) + (scc "sum" sum xs)
  ```

  even though one of the annotated expressions may evaluate the spine of `xs`, the cost of evaluating `xs` is excluded from `"last"` and `"sum"`. Instead, the cost of evaluating `xs` is attributed to the responsible scope for constructing `xs`. This way, the reported costs of expressions are abstracted away from the evaluation degree of their free variables.

- By default, *constant applicative forms* (CAFs) are automatically annotated by the compiler with a special `"CAF"` cost centre. Since CAFs are evaluated only once (if needed), several expressions may be responsible for different parts of the evaluation. Instead of attributing the costs of evaluating CAFs to the demanding expression(s), the cost of evaluating a CAF is attributed to its corresponding `"CAF"` cost centre. Non-updatable CAFs are treated equally.

- The evaluation costs of unprofiled expressions are *subsumed* by the cost centre which *references* the unprofiled expression. If `f` is an unprofiled function in

  ```
  scc "funMap" map (f y) xs
  ```

  then the cost of evaluating `f` is attributed to the referencing cost centre `"funMap"`. This form of inheritance allows programmers to aggregate the costs of heavily used functions (e.g. library functions) to their *reference site*: the source location referring to the function.

To be precise about the rules of cost centre attribution, Sansom in [7] developed a formal notion of cost centre attribution using an operational abstract cost semantics. Basically, a natural semantics for lazy evaluation [58] is extended with the notion of cost centre attribution.

Two abstract cost semantics are described and compared: *lexical scoping* and *evaluation scoping*. In lexical scoping, the cost of evaluating an expression is attributed to the cost centre enclosing the *declaration site*; i.e. enclosing the expression currently being executed. In evaluation scoping, the cost of evaluating an expression is attributed to the cost centre enclosing the expression's *application site*. Take for example[2]

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

Under lexical scoping, the cost of evaluating `(23*16+1)` is attributed to the cost centre `"fun"`. However, using evaluation scoping the cost of evaluating `(23*16+1)` is attributed to the cost centre in scope when `f` is applied.

The differences between lexical scoping and evaluation scoping are subtle. Most notably, unlike evaluation scoping, lexical scoping corresponds to "intuitions" since it reports the costs of expressions with respect to their *reference site* in the source program. Furthermore, the total cost of an expression can be simply measured by annotating the expression. On the other hand, evaluation scoping offers a finer distribution of costs, although measuring the total cost of an expression requires annotating the corresponding application site, which can be tedious and difficult to track. In practice, unlike evaluation scoping, lexical scoping is easier to maintain during the transformation phases of the source program.

Experience gained from both lexical and evaluation scoping suggests that the former is more practical for profiling. Eventually, the lexical attribution scheme was distributed in the GHC profiling system [7][59].

The optimising compiler GHC performs many program transformations to generate efficient code [60]. The target code is based on the Spineless Tagless G-Machine [61]. At compile-time, profiled expressions are maintained during transformations to preserve the scope of cost centres, which is a major undertaking. Moreover, the runtime system is modified to collect profiling information; for instance, the machine state is equipped with a *current cost centre* register and the cost centre corresponding to the expression currently being executed is stored in this register. Furthermore, every heap closure is tagged with a cost centre so when a closure is *entered*, its cost centre is loaded into the current cost centre. Incurred costs are attributed to the cost centre held by the current cost centre register. The

[2]This example is taken from [57].

heap profiler implementation is similar to [24]: at regular intervals, execution is suspended and the entire heap is traversed for a *census*. Profiling results are post-processed to generate charts using the same program (`hp2ps`) by [17][24].

Initially, the heap profiles provided by GHC are based on the heap profiles of HBC/LML [24] and NHC [17]. This involves *producer profile*[3] (including producers of a module or group of modules), *description profile*[4] (including description by type) along with the ordinarily heap profiling restrictions. However, *retainer profile* and *lifetime profile* are not provided.

Compared to the heap profiles of HBC/LML [24] and NHC [17], GHC's heap profiles [7] (when profiling with automatic annotation for top-level functions) are similar though there is a major difference in the producer profile: with the cost centre profiling scheme in GHC, unprofiled functions are subsumed by their reference site. In effect, this gives a finer summary of the overall heap consumption. Furthermore, with the ability to annotate expressions by a user-defined label, the costs of the many instances of (say) `map` can be distinguished since they are attributed to their reference site. In HBC/LML and NHC, however, there is no aggregation method of this sort. Distinguishing the costs of function instances requires writing new functions with different names.

From a heap-profiling perspective, the subsuming technique together with the freedom in selecting expressions is helpful for the reasons mentioned above. Lexical profiling, while effective in practice, suffers from several problems. A major drawback is the special treatment of CAFs: costs are attributed to CAFs instead of to the cost centre of the expression which demands the value of the CAF; this can result in confusion. Indeed, programmers have been complaining about this issue [59]. The second concern regards unprofiled library functions. By default, prelude functions are subsumed; profiling prelude functions requires manually annotating every function, which can be tedious.

In subsequent developments, GHC adopted two additional heap profiles [62]: *retainer profile* [17] and *biographical profile* [39]. Unlike NHC [51], profiling restrictions between biographical and retainer profiles are prohibited [36][54], which is a drawback since this restriction can be useful in pinpointing space leaks.

Further improvements to the GHC profiler involved the support of a *cost centre stack* [62]. A cost centre stack extends the idea of a cost centre (under lexical scoping) with *statistical inheritance* of cost centres for *time processing* [8][63]. This is similar to call graphs in a call-by-value environment. Cost centre stacks are extended in GHC to support time and space allocation, excluding heap profiles [54]. Instead, call graphs appear as a part of keys in producer profiles. However, keys with many call chains do not fit in heap profiles and are therefore unreadable.

2.4 GHC Profiling

In this section we discuss current heap profiles using GHC, a current state-of-the-art compiler for Haskell. Section 2.4.1 demonstrates types of heap profiles. Section 2.4.2 briefly discusses the relation of heap profiles to space leaks.

[3]Treated as cost centres.

[4]Description profile is termed *construction profile* in [1][17][24].

2.4.1 Current heap profiles examples in GHC

We demonstrate four kinds of heap profiles: *producer*, *construction*, *retainer* and *biographical* profiles. Heap profiles are demonstrated using GHC. This compiler adopted many, though not all, methods developed for the original implementation of heap profiles in [1][16]. There are several different implementation details between the original heap profiles and GHC. However, they are broadly similar.

The program for which we provide heap profile examples is in Figure 2.3. This program takes a natural number nat as its input, constructs a list of natural numbers ranging from 1 to nat, sums the elements of a list and adds the first member of a list to the final result.

```
mkList x y
  | (x == y)  = [x]
  | (x <  y)  = x : (mkList (x+1) y)

sumWith v []     = v
sumWith v (x:xs) = sumWith (v+x) xs

sumsList' xs = sumWith 0 xs + head xs

sumsList nat = sumsList' (mkList 1 nat)

main = do
       nat <- getLine
       let nat' = read nat :: Integer
       print (sumsList nat')
```

Figure 2.3: The sumsList program.

In GHC, heap profiles are generated in three stages. First, the program is *compiled* with a runtime system option to generate profiling information. The program is then *executed* with a heap profiling option, each option corresponding to a different type of heap profile. During the execution of the program, heap profiling information is regularly collected and logged to a “.hp” file. However, the profiling data contained in a “.hp” file are not intended for human readers. The next step is to *transform* the “.hp” file to a graphical chart through a separate program named HP2PS [64] that creates a PostScript[5] file [54].

We start with a simple kind of heap profile. The *producer profile* of the sumsList program is shown in Figure 2.4. The shape of the graph describes how the volume of *live heap memory* changes over the execution time of the program. Live heap memory is represented by the vertical axis and execution time is represented by the horizontal axis. Overall, the shape of this graph demonstrates two phases: the live heap grows dramatically, reaching a peak of about 35 MB, and then begins to decline.

The list of keys shown to the right of the graph are associated with shaded bands in terms of the source program: each key corresponds to a different program

[5]PostScript is a registered trademark of Adobe Systems Incorporated.

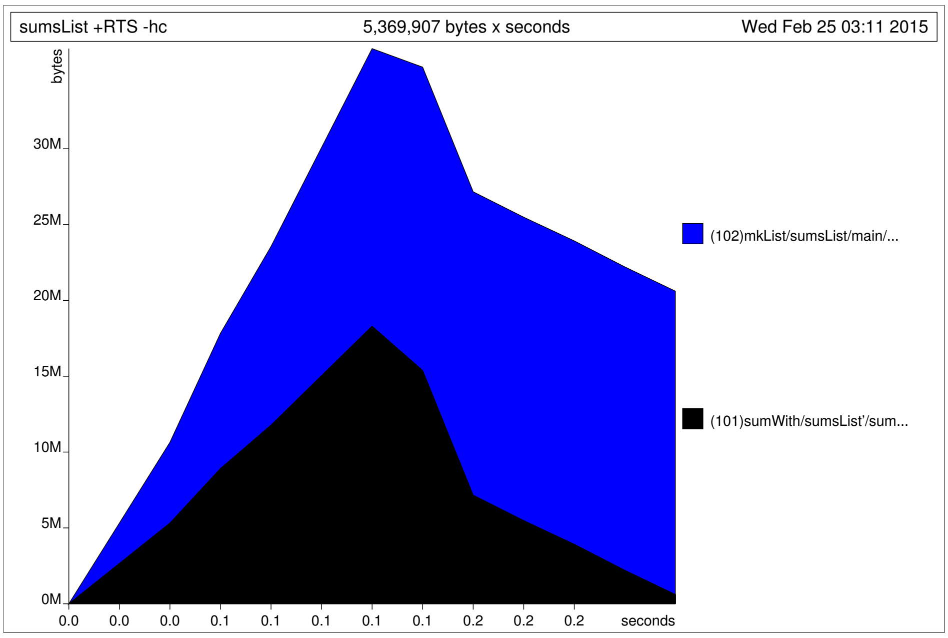


Figure 2.4: A producer profile of `sumsList`.

component. Furthermore, function names separated by the forward slash / in keys represent a *call-graph* of a function. For instance, the function `mkList` is called by `sumsList`, which is in turn called by `main`. For the `sumsList` program, this producer profile shows that a large amount of heap is produced by the function `mkList`. By default, 20 is the maximum number of bands in a profile. Program components occupying less than 1% of the overall heap space are omitted from the profile. For large programs, the number of bands can be large. In such cases a program can be profiled by producers of *modules* to accommodate heap bands in a profile [24][54].

At the top of the heap profile is a title containing three elements: the name of the program together with the heap profiling options, the total volume of the graph and the date on which the program is executed.

The *construction profile* of the `sumsList` program is shown in Figure 2.5. Here, the keys marked as `:` and `S#` are constructors. The former represents `Cons` cells, while the latter represents numbers of type `Integer`[6]. In the context of `sumsList`, the heap band associated with the keys `:` and `S#` represents a list of integers.

Besides constructors, function names in a key can refer to a *closure*, which is an unevaluated expression [36]. For instance, in Figure 2.5, a large section of the heap is associated with `<main:Main.sat_sIL>`. In terms of the source program, this key identifier refers to the unevaluated expression `(v+x)` in the second equation of `sumWith`[7].

A concise form of construction profile can be obtained by using a *type* profile [24][54]. This is useful for programs with large numbers of constructors. The `sumsList` program is simple so a type profile is not necessary here.

Note the similar shape of graphs between the producer profile in Figure 2.4 and the construction profile in Figure 2.5. The slight variation in the shape of the

[6]The key `S#` represents a GHC library-defined constructor [65].

[7]The identifier `sIL` is a compiler-generated *let-binding*.

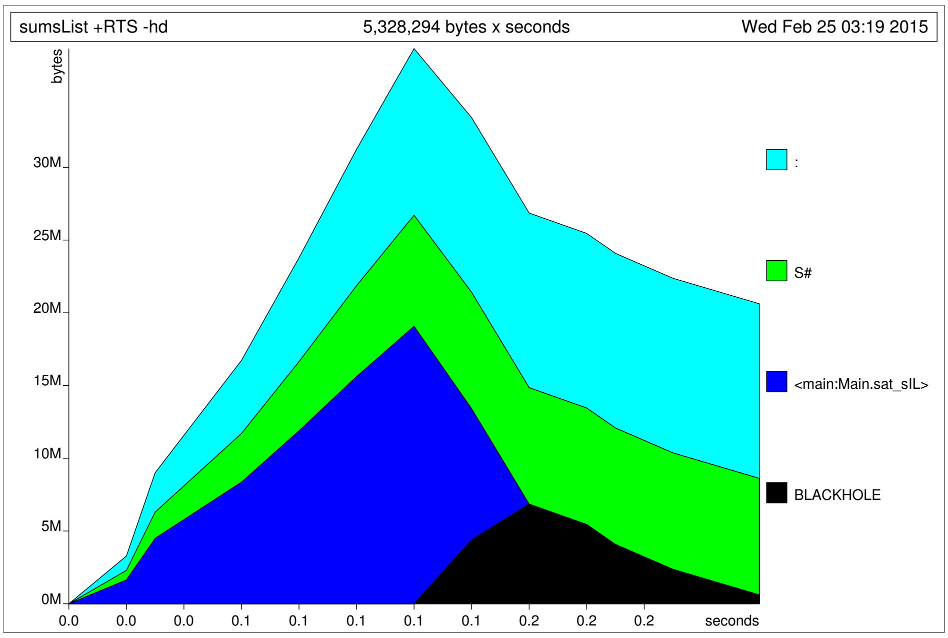


Figure 2.5: A construction profile of `sumsList`.

graphs is due to the time-sampling profiling scheme.

The next profile is the *retainer profile*. The retainer profile of `sumsList` is shown in Figure 2.6, with the functions `sumsList'` and `sumWith` shown as retaining the majority of heap memory cells. Program components kept in the heap by many retainers are represented by a *retainer set*. To reduce profiling overheads, the number of retainers in a set is fixed to a limit (eight in GHC). Moreover, retainers exceeding a specified set limit are collapsed into a `MANY` key [54].

There are different cases where retention can occur. An example is unevaluated expressions holding onto heap cells until they are completely evaluated. Another common instance is a *constant applicative form* representing a data structure [36]. Heap memory can also be retained by the compiler *system stack*. In GHC, the system stack is identified by `SYSTEM` as shown in Figure 2.6 [54].

Finally, there is the *biographical profile* in Figure 2.7. The ultimate goal of biographical profiling is to identify "*live but useless*" heap cells [39].

In Figure 2.7, heap bands are associated with use, lag and void keys. A large fraction of heap space seems to be used. As a general rule, heap memory is used if computations are evaluated. The space associated with void suggests a minor heap waste. There is an apparent lag which suggests accumulation of suspended computations from the start of the computations until about half the time of execution.

Unlike the other heap profiles, a biographical profile does not relate heap sections to the source program; keys are marked with the four biographical classes of cells. However, with the support of profiling options one can restrict different parts of the heap to other profiles (e.g. producers of lag or drag). What distinguishes the biographical profile from other profiles is the ability to suggest a space leak directly. Examples for biographical profiling can be found in [36][39].

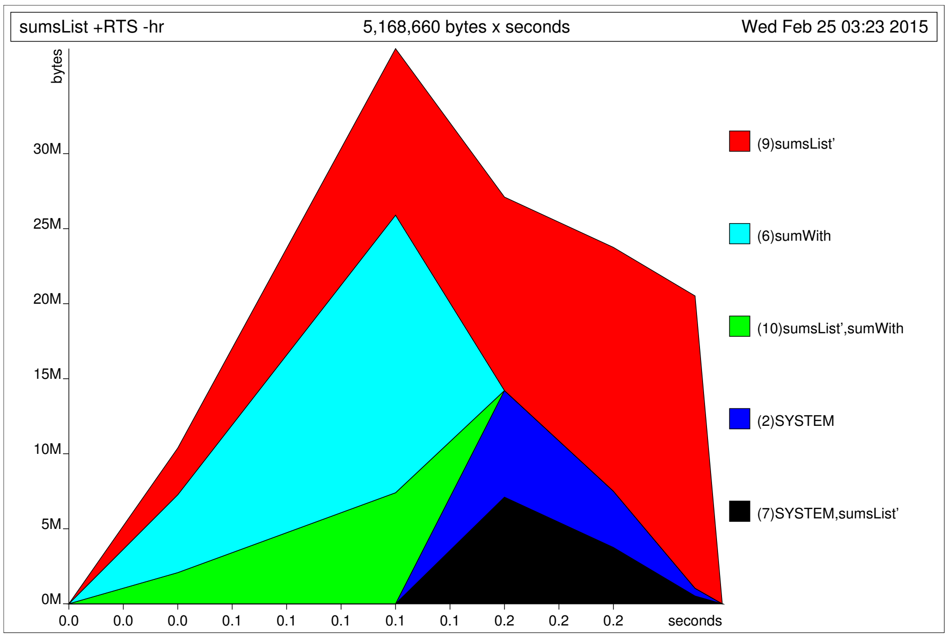


Figure 2.6: A retainer profile of `sumsList`.

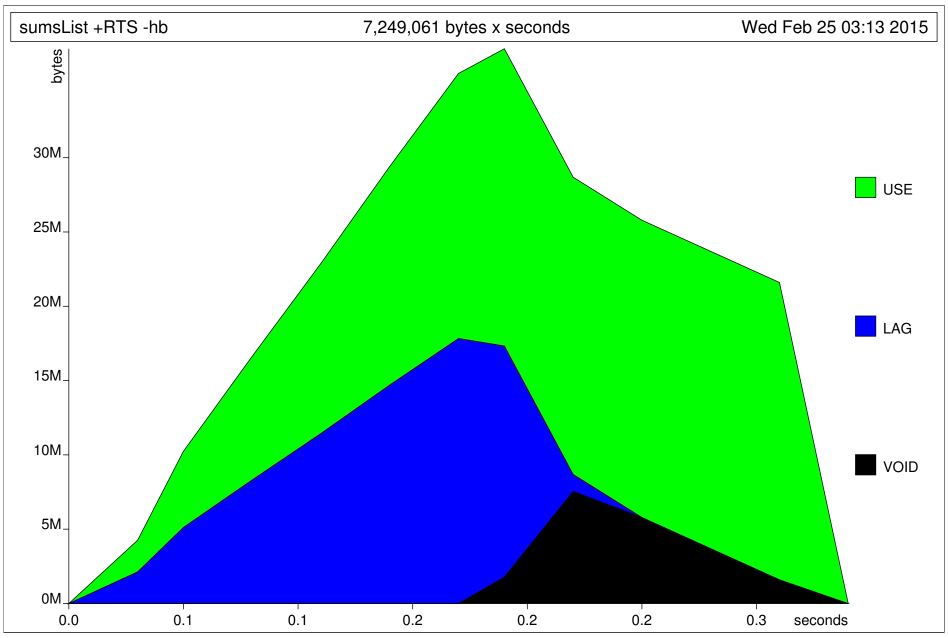


Figure 2.7: A biographical profile of `sumsList`.

2.4.2 Relation to space leaks

Heap profiling is one way of studying the space behaviour of programs. Resorting to different kinds of heap profiles, one can diagnose a space leak. Which type of heap profile directly helps in fixing a space leak?

A common technique is to start by examining a biographical profile since this profile gives a good image of the overall heap usage. The biographical classifications can suggest some instances of space leaks at a high level. For instance, a *drag* band indicates wasted heap space. The offending heap section can be further examined at the source-level of the program by combined profile *restrictions* [36][39].

2.5 Discussion

There is a large body of work on heap profiling. The work by Runciman and Wakeling [24] and Runciman and Röjemo [17][39] has received some attention from those working on other programming languages. For example, Novark, Berger and Zorn introduced HOUND, a tool for detecting C and C++ memory leaks. They use the idea of drag to detect memory leaks caused by *objects* with long dragging time [66]. Shaham, Kolodner and Sagiv reduced the drag amount of objects in Java programs [67]. Nilsson developed retainer profiles in HEAPY, a memory profiler and debugger for Python [68]. Hallenberg designed a heap profiling system for Standard ML with region-based memory management. This profiler shows memory use for *regions* [69].

In the object-oriented languages community, the classifications provided by the biographical profile [39] are seemingly used to increase the degree of automated support for reducing space consumption using profiling information.

Memory profiles of functional implementations (e.g. producer, construction, retainer and biographical) provide various kinds of valuable information. However, there are two drawbacks:

- Relating heap profiling information to the source program at the expression-level can be difficult. The heap profiles by Runciman and Wakeling [24] and Runciman and Röjemo [17][39] provide a reference to an entire body of a function. The cost centre profiling scheme by Sansom [7] equipped in GHC allows the user to annotate expressions of interest explicitly with distinct cost centres. However, annotating programs can be a tedious task that sometimes requires changing the structure of expressions. One possible way of relating expressions to heap profiling results automatically is to profile constructions by occurrence.

- Space leaks causing stack overflow are common in functional programs due to their recursive nature. Stack profiling has not received enough attention in lazy functional implementations. The GHC heap profiling system provides a profiling option to include stack memory in several kinds of heap profiles. However, stack memory is seemingly treated as heap memory: space occupied by the stack is aggregated to a single undifferentiated category in memory[8] [54]. In a similar manner to heap profiling, the stack memory can be viewed in different ways (analogous to producer, construction and biographical views of heap memory).

We therefore propose two kinds of new memory profiles. The first is a *hotspot* profiler which automatically annotates expressions at the *occurrence-level* (Chapter 4). The second is a *stack profiler* supporting *producer* and *construction* profiles (Chapter 5).

[8]This information is gathered based on the GHC profiling documentation in [54] and by experimenting with profiling options and results using recent versions of GHC-7.8.

Chapter 3

Pure: A Lazy Functional Language Implementation

We have implemented a compiler for a pure lazy functional language we call *Pure*. This chapter introduces the implementation of Pure for which we implement memory profilers in subsequent chapters. Section 3.1 gives an overview of the Pure implementation. Section 3.2 describes *Core*, the intermediate language to which Pure is compiled. Section 3.3 concludes with a brief demonstration of the underlying abstract machine of Core, the *G-machine*.

3.1 *Pure* Implementation

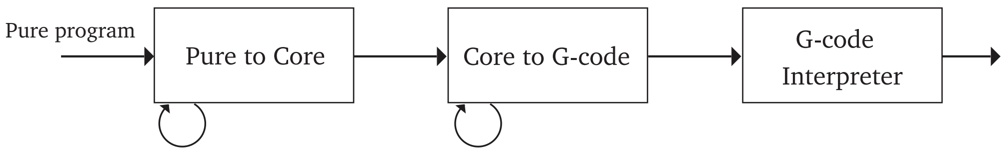


Figure 3.1: Pure implementation phases.

We have implemented a compiler for a pure first-order lazy functional language we call *Pure*. The Pure language is a minimal subset of Haskell supporting various syntactic constructs and features found in a typical lazy functional language including *let/letrec* constructs, *if* and *case* expressions, *pattern matching* and *guards*. Algebraic data type declarations are supported by the Pure language, with built-in lists and booleans, where lists are constructed with `Cons` and `Nil` *constructors*. The only literals are integers. Below is an example of a function that sums the elements of a given list written in the Pure language.

```
sum Nil         = 0;
sum (Cons x xs) = x + sum xs;
```

As with conventional compilers, compiling and executing a Pure source program proceeds in a series of phases, the output of each phase becoming the input of the following phase. Figure 3.1 illustrates the overall structure of the phases for the implementation of Pure. The first two phases include the compiler proper whilst the third phase represents the runtime system. The former is written in

Haskell whereas the latter is written in C. Except for the runtime system, each phase includes a number of passes mostly concerning syntax-tree transformations and translations. Below, a brief description of these phases follows.

A parser reads a Pure program and constructs a Pure abstract syntax tree that is subsequently transformed to a simpler Pure representation by removing syntactic sugar. In particular, a desugaring pass transforms pattern matching, guards and `if` expressions to (possibly nested) *simple case expressions*, translating Pure to a program in a language called *Core*. The Core language is a version of Pure with simpler constructs which serves as an intermediate language suitable for code generation. After a number of Core-related administrative passes (e.g. annotating Core with information requested in code generation such as labelling algebraic data types), Core is translated to *G-code* ready for execution by a *G-machine* [50].

The Core language and the G-machine are of most interest since they are used to explain the implementation of our memory profiling tools (in Chapters 4 and 5). Section 3.2 briefly describes the Core language and Section 3.3 describes the Core G-machine.

3.2 The *Core* Language

The *Core* language is essentially a desugared (or simpler) version of the Pure language with complex constructs (e.g. pattern matching) transformed to simpler constructs. Core is intended to be simple so that it is possible to transform Pure (and similarly rich functional languages) into Core without loss of expressiveness. As with conventional functional implementations, Core provides an intermediate interface between high-level constructs and a low-level representation of compiled code. The syntax of Core is given in Figure 3.2.

A program in Core is a list of top-level function definitions delimited by semicolons. Every function is defined by a sole *equation*. Function arguments are simple variables, so pattern matching is not permitted. Furthermore, zero-arity definitions are called CAFs, though they are treated as functions.

Function definitions can contain local bindings in `let` constructs and recursive local bindings within `letrec` constructs. The left-hand side of a binding in a `let` or `letrec` expression is restricted to a simple variable.

As usual, *constructors* are included in the language where each constructor is identified by a unique label and assumed to be *saturated*, that is, applied to the exact number of its arguments. Although not explicitly shown in the Core syntax, built-in algebraic data types are included; in particular, lists and booleans are desugared from the Pure language (see Section 3.1).

The final construct is that of a `case` expression which performs pattern matching. A `case` construct consists of expression e to be analysed and an alternative list. Every alternative consists of a constructor (identified by a label) followed by a number of *simple* variables which must be equal to the arity of the corresponding constructor and an expression e to the right-side of the arrow.

Finally, there are binary arithmetic and comparison operators (e.g. `+`, `<` *et cetera*), and integer literals.

Program	$program$	$\rightarrow$	$fun_1 \; ; \ldots ; \; fun_n$	$n \geq 1$
Functions	fun	$\rightarrow$	$f \; v_1 \ldots v_n = e$	$n \geq 0$
Expressions	e	$\rightarrow$	$e \; e$	Function application
		$\mid$	$e \odot e$	Primitive operation
		$\mid$	`let` $binds$ `in` e	Local bindings
		$\mid$	`letrec` $binds$ `in` e	Local recursive bindings
		$\mid$	`case` e `of` $alts$	Case expression
		$\mid$	$C \; e_1 \ldots e_n$	Constructor $n \geq 0$
		$\mid$	$atom$	Atomic expression
Atoms	$atom$	$\rightarrow$	v	Variable
		$\mid$	i	Integer
Bindings	$binds$	$\rightarrow$	$bind_1 \; ; \ldots ; \; bind_n$	$n \geq 1$
	$bind$	$\rightarrow$	$v = e$	
Alternatives	$alts$	$\rightarrow$	$alt_1 \; ; \ldots ; \; alt_n$	$n \geq 1$
	alt	$\rightarrow$	$C \; v_1 \ldots v_n \rightarrow e$	$n \geq 0$
Variables	v, f			

Figure 3.2: Syntax of the *Core* language.

In essence, this simple language is the Core language of [70, Chapter 1] restricted to first-order programs. This adaptation facilitates the process of targeting the G-machine (see Section 3.3).

3.3 The *Core* G-Machine

The *G-machine* is a stack-based abstract machine for executing lazy functional languages by way of *lazy graph reduction*, originally developed by Augustsson and Johnsson [50][71]. The G-machine implementation of Core is thoroughly described in [70, Chapter 3]. Here follows a brief overview of the G-machine.

In a typical G-machine, each top-level function[1] definition is compiled into a sequence of G-code. These instructions are used as graph rewrite rules to reduce the graph of a right-hand side, corresponding to a function body, to a value.
Graph reduction is carried out by rewriting the left-hand side of a function definition to an instance of a corresponding right-hand side with argument *pointers* substituted for formal parameters, *updating* the root of the left-hand side graph with the root of the right-hand side instance, thereby preserving *laziness*. This

[1]Resembling a *supercombinator* [11, Chapter 13].

cycle is repeated until the graph is eventually reduced to a *weak head normal form* (WHNF).

For a simple example of graph reduction by the G-machine, consider the function definition `square x = x * x` compiled to the following G-machine code[2].

```
square:
   PUSH 0
   PUSH 1
   PUSHFUN *
   MKAP
   MKAP
   UPDATE 1
   POP 1
   UNWIND
```

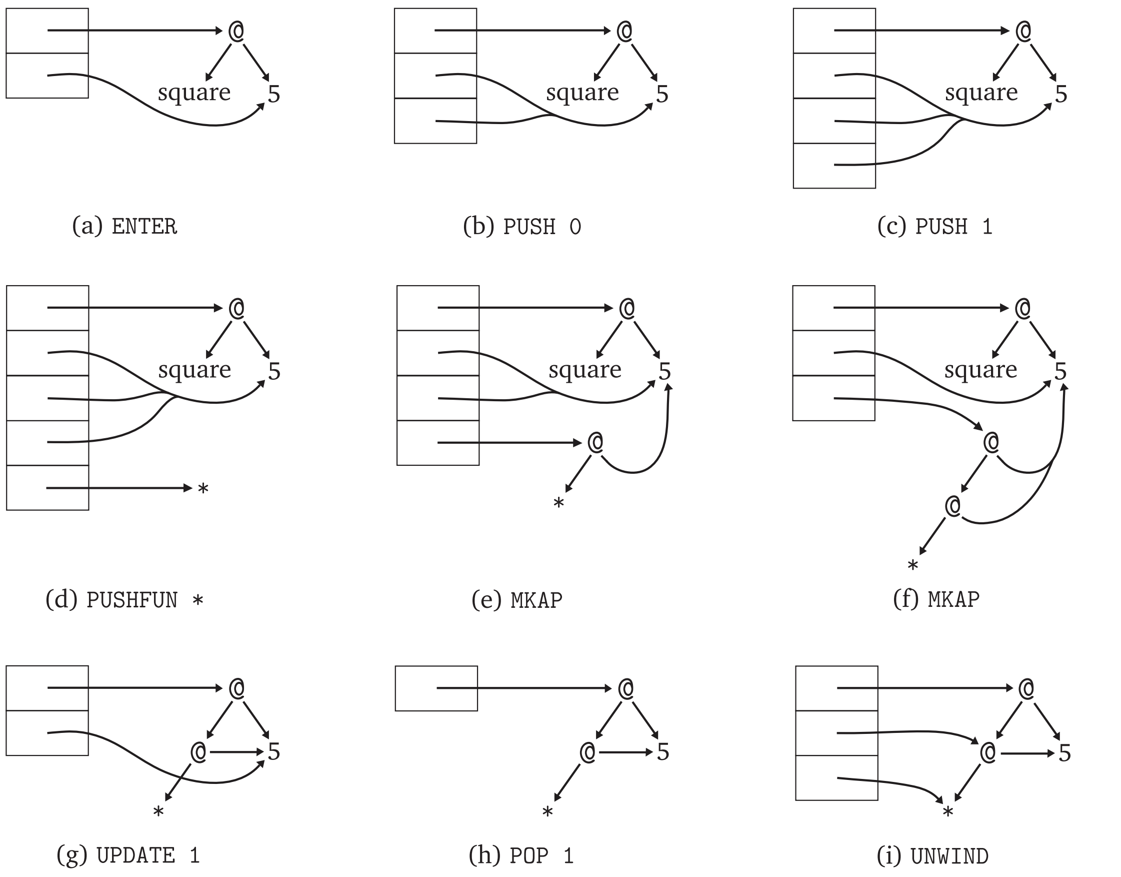


Figure 3.3: G-machine reduction of `(square 5)`.

Figure 3.3 demonstrates G-machine execution of rewriting `(square 5)` to the right-hand side of `square` definition `(5 * 5)` (i.e. reduction steps from (a) to (h)). The symbol `@` denotes an application node and arrows are pointers from the stack to heap nodes.

Figure 3.3(a) shows the stack configuration when the expression `(square 5)` is ready for reduction after the spine of the graph `(square 5)` has been unwound

[2]Note that we have used a simplified code for demonstration purposes.

and the stack is *rearranged* in order to access the argument (5) of `square` through the topmost pointer of the stack (the stack grows downwards); the second topmost pointer pointing to the root of the graph of `(square 5)` is retained to be updated by the right-hand side instance of `(square 5)`. The G-machine now ENTERs (i.e. jumps to) the code of `square` and starts executing its code. The PUSH n instruction (in (b)-(c) of Figure 3.3) pushes the nth element of the stack *relative to the top of the stack* with the topmost stack pointer starting at offset zero. The instruction PUSHFUN * pushes a pointer to a function node representing the multiplication `*`. The MKAP instruction (in (e)-(f) of Figure 3.3) allocates an application node in the heap with the two topmost pointers of the stack as branches of the application node; note how the arguments to the function `*` are shared. Having the graph corresponding to the right-hand side of the definition of `square` constructed, the root node of the original application `(square 5)` is updated (i.e. overwritten) with an *indirection node* (not shown in Figure 3.3(g)) pointing to the root node of the new graph, and the unneeded arguments of `square` are popped off the stack by the instructions UPDATE 1 and POP 1, respectively (Figure 3.3(g)-(h)), thereby having the graph of `(square 5)` rewritten to `(5 * 5)`. The last instruction of `square` causes the G-machine to enter the UNWIND state to continue reduction by unwinding the spine of the new graph on top of the stack to find the function node `*` of the application `(5 * 5)`, pushing pointers to the application nodes along the way (Figure 3.3(i)). The stack is then rearranged and the code for the function `*` is entered, reducing `(5 * 5)` to `25`.

The complete semantics of the Core G-machine are provided in Appendix A. Section A.1 contains the compilation of Core to G-code defined by a set of *compilation schemes*. Furthermore, the operational semantics of the instructions used by the compilation schemes are defined by the *state transition rules* of the G-machine, which are provided in Section A.2. Note that the Core G-machine is a basic version and does not incorporate many optimisations, so it may be prone to space leaks.

Chapter 4

A Hotspot Profiler

In this chapter we introduce a *hotspot* profiler: a new kind of heap profile which highlights hotspots by *occurrence* in programs. Section 4.1 describes the idea of hotspot profiling. Section 4.2 explains our implementation of a hotspot profiler. Section 4.3 demonstrates the application of hotspot profiling to a series of programs.

4.1 Overview

In this section we describe the main idea behind profiling *hotspots* by *occurrence*. In Section 4.1.1, we explain what we mean by occurrences and the motivation for occurrence profiling. Section 4.1.2 discusses the design goals and the key ideas that led to the construction of a hotspot profiler. Section 4.1.3 describes the details of hotspot profiling including the information that it represents.

4.1.1 Occurrences

Current heap profilers for lazy functional languages (e.g. [7][16]) provide various kinds of information about memory, which are generally effective in examining the way programs use heap space. Memory populations are aggregated to provide concise summaries. In particular, constructions of the same value are aggregated and referenced by a corresponding top-level function. While this technique supplies succinct information, the results can lack the (necessary) precision to locate *automatically* a dominant source of space consumption in the program: occurrences of construction symbols are referenced to the entire body of a function whereas distinct locations corresponding to individual symbolic occurrences can be more informative. Take, for example, the definition in Figure 4.1.

```
maxcc mos m Nil = mos
maxcc mos m (Cons n ns)
  | n >  m = maxcc (Cons (Cons n Nil) Nil) n ns
  | n == m = maxcc (Cons (Cons n Nil) mos) m ns
  | n <  m = maxcc mos                     m ns
```

Figure 4.1: The definition `maxcc` where `Cons` occurrences are annotated.

Here, every occurrence of the construction symbol `Cons` is boxed. Suppose that some `Cons` above impose high sustained-heap pressure in a program. A producer profile *restricted* by `Cons` would report the function `maxcc` as the producer. But which occurrence(s) of `Cons` is to blame?

So there is a gap between profile results and expression-level reasoning. One way of decreasing such a gap is to profile every occurrence. An occurrence is either a 1) function; 2) constructor; or 3) literal. The `maxcc` definition annotated with occurrences is shown in Figure 4.2.

```
maxcc mos m Nil = mos
maxcc mos m (Cons n ns)
   | n >  m = maxcc (Cons (Cons n Nil ) Nil) n ns
   | n == m = maxcc (Cons (Cons n Nil) mos)  m ns
   | n <  m = maxcc mos                      m ns
```

Figure 4.2: The definition `maxcc` where every occurrence is annotated.

At a glance, we see a problem: if every occurrence is profiled, the result can be distorted in the presence of so many occurrences, which is unwieldy. Programmers wishing to improve memory demands are normally interested in pinpointing memory *bottlenecks* or *hotspots* and not in the detail of components consuming minor fractions of space.

We propose a *hotspot profile* which presents concise memory statistics of *hotspots* by source occurrence and relates them to the corresponding locations in a source program.

4.1.2 Design outline

The design options for occurrence profiling are the outcome of a previous attempt and experiences of using heap profiling tools similar to those described in [1][17]. An outline of the primary design decisions are as follows.

Scalability Profiling large programs with many modules is a common exercise. It should be feasible to extend occurrence profiling to support such large programs. Moreover, the ability to combine different views of memory is of particular importance to the diagnosis of space leaks.

Profile Charts The effort devoted to the graphical presentation of profile charts in [1] has proven effective in understanding heap profiles. Heap memory is indeed best explained through graphs. These charts are adapted to provide heap charts of occurrences.

Granularity Reporting the cost of every occurrence is undesirable, leading to unreadable profile charts. Presenting information about every function in a program often produces cluttered profiles. Heap profilers overcome this by ignoring functions that account for small space or by grouping functions to provide a coarser form of profiles, which naturally brings the troublemakers to the surface of a heap chart. A similar technique to classify (or group) occurrences associated with high memory demands is essential for practical profile summaries.

Result Presentation Relating the space use of programs at the occurrence level makes source annotation a convenient target since occurrences are easier to examine in source programs. Profiling information is presented in two forms: annotated source occurrences and profile charts. A scheme to relate source annotation with profile charts is easily accomplished through graphical display.

User-friendly Display Graphical design is a complicated field and it is easy to obscure the display of information. The primary goal is to furnish the programmer with clear and simple graphical presentation, with an easily managed graphical interface.

These criteria lead to the design of a *hotspot profile* following the key ideas:

- Space occupied by occurrences is measured in *heap contribution* which we define to be the *overall cost* of an occurrence over the execution time of a program; that is, the percentage of the (space $\times$ time) area under the heap curve in a profile chart [36].
- Each occurrence is associated with a *temperature* which is the heap contribution percentage of an occurrence over the entire cost of the heap graph. Occurrences associated with a high temperature are termed *hotspots*.
- Hotspots are classified by their heap temperature according to a set of *temperature colours*, where each colour represents a range of predefined temperatures. This colour is used to band heap charts of hotspots and annotate the associated hotspot in the corresponding source program, thereby relating heap charts to occurrences at the source level according to their *overall* space use.
- We provide three temperature colours: yellow $<$ orange $<$ red. Red is associated with the hotspots of highest memory demand. Occurrences with lower temperatures than the yellow colour are collapsed into a union band which is neglected in profile summaries. This provides a means to abstract away occurrences occupying small amounts of space from profiles. Temperature values are supplied by the user, otherwise set to a default. A minimum of 10% is imposed on the yellow temperature to avoid cluttering up profile charts.

In summary, a hotspot profile provides information in two forms: profile summaries in charts and annotated source programs. The profile chart represents heap content distributed by hotspot temperatures. Hotspots are annotated in textual displays of source programs according to their temperature colour in profile charts. Further information about hotspots can be investigated in a separate display.

4.1.3 Profiling technique

The hotspot profiler consists of two components: an instrumented compiler and a graphical program. The instrumented compiler gathers profiling information about *occurrences*. The graphical program *post-processes* profiling information and presents it in a graphical form.

The modified compiler collects occurrence information identifying:

- The *producer* that (immediately) allocated the occurrence in the heap.
- The *construction* that the occurrence represents.
- The *creation time* and *allocation counts* of occurrences.

Producers of occurrences are identified by the *declaration name lexically enclosing the occurrence site*. A declaration name is a top-level function variable; an equation binding variable; a local declaration name within *let* or *letrec* constructs. This ensures that occurrences appearing in multi-scoped declarations (e.g. in nested *let* constructs) and equations are attributed to the producer with respect to the lexical declaration scope. By way of example, below is a program fragment where *expr* is an arbitrary expression containing occurrences. Each expression is annotated with its producer, where the top-level declaration name in each equation is distinguished with a source location.

```
f_1 Nil          = expr_f1
f_2 (Cons x Nil) = let a = expr_a in
                     let b = expr_b in
                       expr_f2
f_3 (Cons x xs)  = expr_f3
```

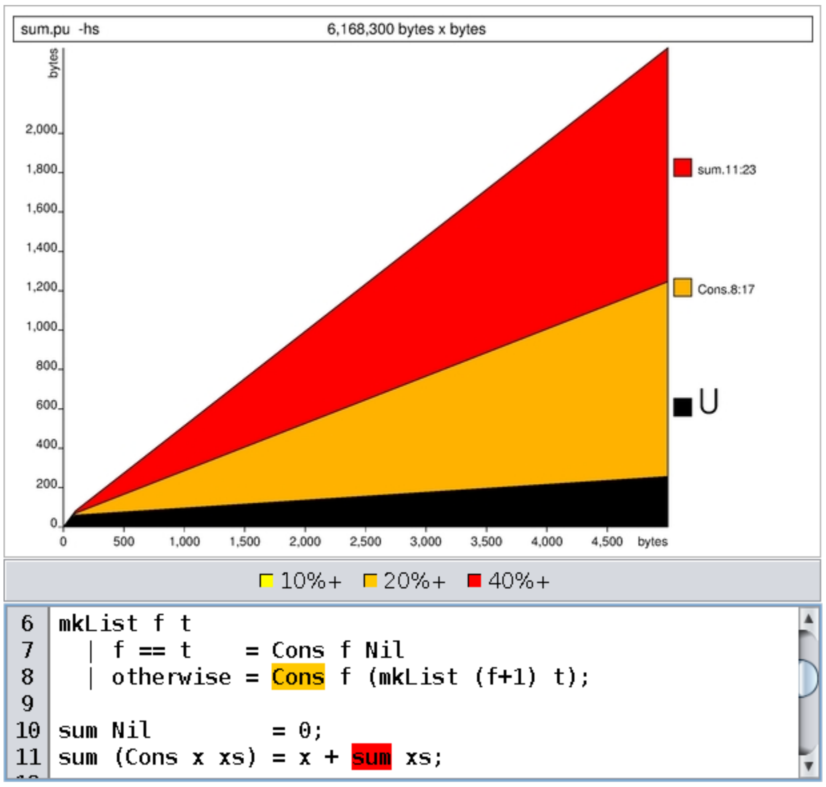
In our implementation the construction of heap cells can take one of the following forms: as of yet unevaluated *closure*, a *constructor* (e.g. `Cons`) or a literal (e.g. `Int`). For closures, we take the function name to serve as the construction representation.

Constructions are used to mark hotspots in sources. However, such static attributes are limited in practice. Information about the dynamic characteristics of hotspots may be essential for comprehending the nature of space leaks. Heap profiles can describe profound hotspot properties when extended with dynamic attributes. For example, suppose that a hotspot profile shows a sustained *plateau* from high demand of memory. Does this indicate a hotspot that is being *retained* or is that hotspot memory being *regenerated* constantly? This is exactly what a *lifetime profile* reveals, which can be derived from heap cells' creation times [17]. In addition to occurrences' creation times, allocation counts of occurrences are collected to serve as supplementary information in a lifetime profile.

Auxiliary information about constructions and producers such as source location is also gathered. This extra information is used by the implementation and proves helpful when multiple hotspots are examined.

When a program is executed for profiling, the implementation suspends the computation and traverses the heap for a *census* at fixed intervals, collecting *population counts* for each occurrence at every distinct creation time. After heap traversal is complete, a census is logged into a file and the computation proceeds. When execution terminates, a supplementary profile table is appended to this file. Amongst other information, log files contain a *three-dimensional* profile of each occurrence at each census: that is, construction, producer and creation time(s).

The second component of the hotspot profiler is a graphical program which post-processes profiling data and generates a hotspot profile in a graphical form. The hotspot profile consists of three parts: a *hotspot-construction* heap profile, a *temperature colour* bar and a command-oriented textual display of the source program which highlights *hotspot occurrences*. This heap profile and the corresponding hotspot in the source program are coloured according to the temperature. A hotspot profile example is shown in Figure 4.3.



```
6  mkList f t
7    | f == t     = Cons f Nil
8    | otherwise = Cons f (mkList (f+1) t);
9
10 sum Nil         = 0;
11 sum (Cons x xs) = x + sum xs;
```

Figure 4.3: A hotspot profile example.

The hotspot-construction graph in the top area of Figure 4.3 shows how the *live heap* size varies over the execution time of the program, where time is calculated by heap *allocation counts*. Live heap memory (vertical axis) and heap allocation counts (horizontal axis) are both measured in bytes. In this example, the live heap reaches a peak of about 2200 bytes whereas heap allocation counts are slightly above 4500 bytes. The latter can be taken as an *approximation* of *total heap allocations*. Interpolation is linearly employed between *censuses*, which could deceive the programmer (e.g. by failing to capture heap spikes). More accuracy is gained by increasing census frequency.

The title on the top area of the heap profile contains the name of the profiled program, the executed command when running the graphical interface and the *overall cost* of the program in (live heap bytes $\times$ heap allocation bytes).

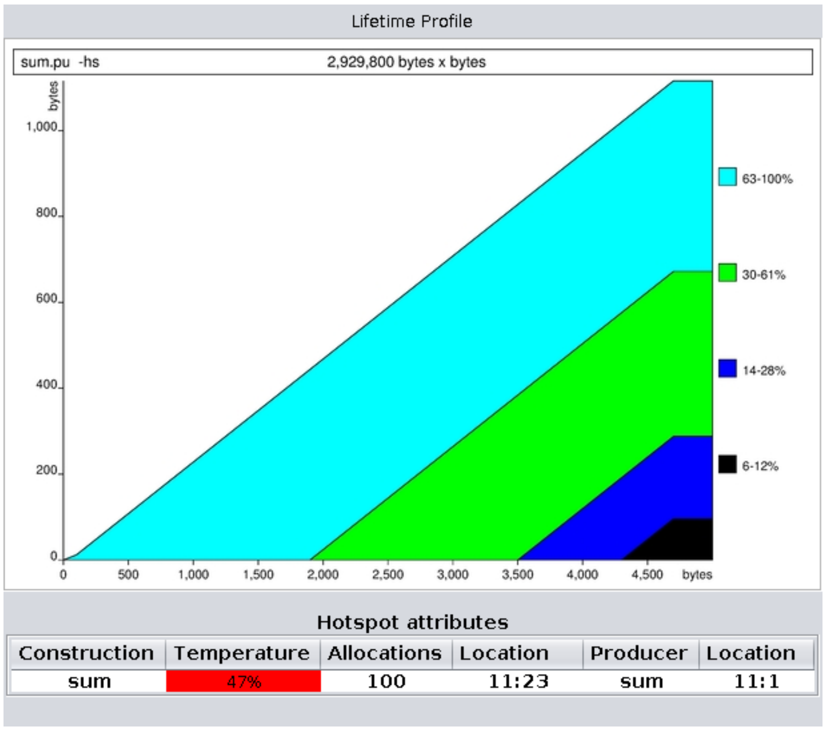
This heap chart is distributed by the *temperature* of hotspot construction according to the *temperature colour* bar shown below the graph. The temperature of an occurrence is calculated by the contribution amount of the area under the (bytes $\times$ bytes) heap curve to the total area of the heap graph. Temperature colours are specified by a lower percentage. There are three temperature colours, characterised by yellow, orange and red — this naturally resembles "heat" in increasing order. Occurrences belonging to the yellow, orange or red temperature are termed *hotspots*.

Every band in a hotspot-construction chart is associated with an identifier shown in the key set to the right. A key of the form *identifer.row:column* represents a hotspot construction followed by its source location. As the reader will have noticed, each band is coloured in accordance with the specified temperature. Heap bands associated with a temperature smaller than the yellow coloured (those with heap contribution of less than 10% in this example) are collapsed into a union band (U). As we shall see in Section 4.3, heap bands can have the same colour if hotspots are of the same temperature.

In essence, this heap graph is the construction profile in [1], revised to show hotspots.

Finally, there is the command-oriented textual display of a source program at the bottom of the hotspot profile example. Hotspot constructions are annotated with respect to their temperature colour. With the aid of heat-marked source code and the heap graph, one can locate hotspots in the source program and study the corresponding heap band. This combination can be regarded as a first hotspot resort — to be examined in more detail.

In this example, there are two hotspots: `sum` belonging to the red temperature with a heap contribution above 40% and `Cons` associated with the orange temperature with a heap temperature above 20%.



Hotspot attributes

Construction	Temperature	Allocations	Location	Producer	Location
sum	47%	100	11:23	sum	11:1

Figure 4.4: Lifetime profile of an individual hotspot.

Hotspots are investigated further through their individual profiles which are obtained by moving the cursor to a hotspot of interest and pressing the return key. As shown in Figure 4.2, an individual hotspot profile has two parts. The first part is a *lifetime profile* showing two kinds of information: a consistent view and the lifetime distribution of a hotspot-heap graph. For lifetime profiles, the colour of the bands is arbitrary. Similar to the lifetime profile in [17], banding of heap cells corresponds to their *eventual lifetime*. However, for readability, keys are labelled with a percentage range instead of literal lifetimes. The percentage range represents the lifetime of heap cells over the lifetime of the program. For example,

the label 63-100% is attached to heap cells that lived for between 63% and 100% of the total program execution time.
The second part is a hotspot table containing auxiliary information such as the temperature percentage rounded to a whole number and allocation counts. Construction, producer and their source locations are included mainly to assist the process of relating individual profiles to the program component at the source level.

The design of this hotspot profile is motivated by the goal of delivering concise hotspot summaries in a clear graphical representation. Originally, hotspots of the same temperature colour were aggregated in a single heap band so every band in the heap graph was guaranteed to have a unique colour. However, this proved to be unhelpful since we can hardly reason about the graph of a hotspot. The alternative is to band the graph for each hotspot. In consequence, bands of the same temperature class have the same colour. Under the current hotspot scheme, hotspots are generally few in number so heap bands of the same colour are easily related to key entries by looking at both in the same order.

The other issue concerns occurrences that cannot be *directly* annotated in the source program, in particular occurrences produced by primitive functions such as integers and boolean results. This is resolved by attributing the cost of results to the primitive function that produced them. The task is then to represent this attribution to the programmer in a meaningful way. One option is to introduce three source annotation marks to primitive functions indicating whether it is 1) a closure without results; 2) results only; 3) closure and results. The other option is to include results in hotspot individual profiles. We take the latter approach to avoid cluttering up the source program with additional annotations.

The notion of temperature colours naturally classifies occurrences according to their space use. By default, the temperature colours are set to `10%+`, `20%+` and `40%+` for the yellow, orange and red temperatures, respectively. The yellow temperature with a minimum of 10% abstracts away (the many instances of) occurrences associated with only small amounts of space and ensures that heap bands can fit a profile chart, and draws attention to occurrences occupying large amounts of heap-space.

In our experience, and as illustrated in Section 4.3, the default temperatures provide a good distribution of hotspots. The user is free to change the temperature percentages otherwise. However, even with modified temperatures, some programs have an *even distribution of heap space,* in which case there are no hotspots.

4.2 Implementation

The purpose of this section is to describe our implementation of a *hotspot* profiler. We start by a brief overview of the modified components and the additions to our implementation for hotspot profiling support. Sections 4.2.2 and 4.2.3 describe the instrumented compiler and runtime system that generate profiling information, respectively. Throughout, we assume that the reader is familiar with *G-machine*-based implementations and the idea of *graph reduction*. For those without such a background, excellent tutorial references can be found in [11][70]. Section 4.2.4 briefly describes a graphical program which displays profiling information.

4.2.1 Overview

We have implemented a hotspot profiler for the *Pure* language implementation (see Chapter 3). The overall modification to the Pure compiler for profiling support is shown in Figure 4.5.

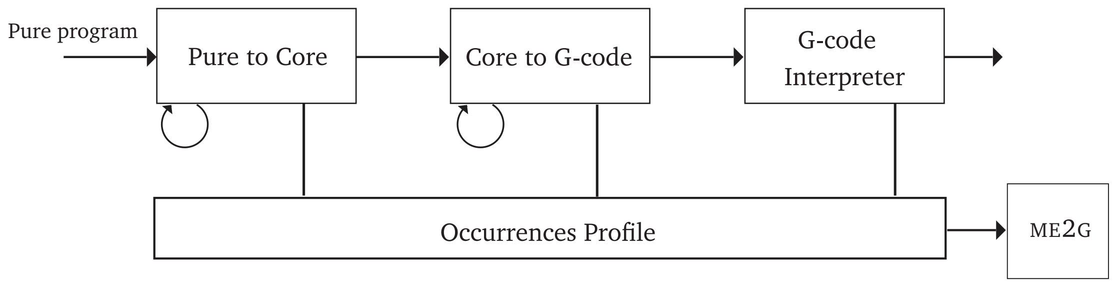


Figure 4.5: Modified compiler for occurrence profiling.

The compiler is instrumented to annotate every occurrence in the source program with profile information. This annotation is preserved throughout the compiler phases: during the desugaring pass of Pure, Pure to *Core* translation and *G-code* generation. The runtime system for the G-code interpreter is instrumented to cope with profiling information and extended for a heap profiling scheme. Meanwhile, profile information about occurrences is accessed and maintained through an occurrence profile structure. Occurrence-heap profiles produced by the runtime system are post-processed by a graphical program (ME2G) which generates a hotspot profile.

4.2.2 Compiler instrumentation

The compiler is modified to gather static information about source occurrences. This information is preserved during compilation and carried to the runtime system. Although straightforward, the modifications to the compiler are numerous and so we describe the main compiler instrumentations only in outline.

Every occurrence appearing within expressions in a source program is associated with an *occurrence tag* using a pair. The pair is an annotation construct of the form $\langle \boldsymbol{\tau}, e\rangle$ which is introduced into the *Pure* and *Core* languages (see Figure 4.6), where $\boldsymbol{\tau}$ is an occurrence tag and e is the associated occurrence. This tag is used to identify occurrences and associate them with profiling information.

Expressions	e	$\rightarrow$	$\dots$	(as before)
		$\mid$	$\langle \boldsymbol{\tau}, e\rangle$	Annotated expression
Profile tag	$\boldsymbol{\tau}$	$\rightarrow$	integer	

Figure 4.6: Annotation construct in *Core* (similarly for *Pure*).

At an early stage in the compilation, the modified compiler annotates every occurrence node in a Pure source tree with a fresh tag τ. In our implementation, occurrence nodes are of three types: a literal node, a construction node or a function node. Application nodes are annotated with the tag of the function they represent; that is, every node in the left-branch chain of a tree is annotated with an identical tag. This allows the runtime system to attribute the space cost of application nodes to the associated function node.

An occurrence tag τ is actually an integer key which maps annotated occurrences to a record in a profile symbol table. This table contains *static* information about occurrences which are determined and added to the symbol table during the traversal of a source tree. The table is retained during compilation and used by the runtime system. Occurrence information includes construction, producer and their source locations.

By way of example, Figure 4.7 shows how the abstract syntax tree of the expression `(sum (sum 50 60) 90)` is annotated with arbitrarily chosen occurrence tags. (As usual `@` denotes application node).

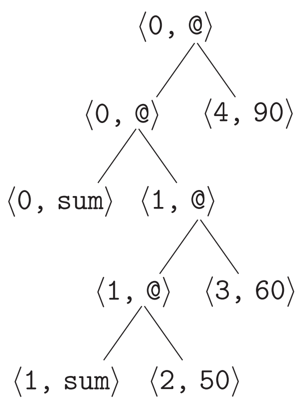


Figure 4.7: Occurrences annotation example in an abstract syntax tree.

Following the occurrences annotation pass, the compiler performs successive transformations mainly involving a set of *syntactic transformations* on a Pure source tree to remove syntax sugar (e.g. compiling *pattern matching* and *guards*). During the transformation process, the implementation preserves the occurrence annotations while complex expressions are replaced by simpler ones. This annotation is then maintained after Pure is translated to the Core language. Since every occurrence is paired with a tag, preserving the pair poses no major difficulties although it requires every transformation to be modified.

The final phase in the compiler translates *Core* to *G-code* instructions. In unprofiled settings, code is generated according to the *compilation schemes* that are described in Section A.1 of Appendix A. For profiled programs, we instrument the compilation schemes to produce G-code instructions annotated with the occurrence tags encountered in Core; every instruction responsible for allocating a heap cell (to be profiled at runtime) is annotated. Annotations for instructions producing *indirection* nodes can be avoided since they are *short-circuited* during garbage collection at runtime. Instructions causing heap allocation are contained in the $\mathcal{E}[\![\ ]\!]$ and $\mathcal{C}[\![\ ]\!]$ rules. The instrumented rules for $\mathcal{E}[\![\ ]\!]$ and $\mathcal{C}[\![\ ]\!]$ are shown in Figure 4.8 and Figure 4.9, respectively.

$$
\begin{array}{lcl}
\mathcal{E}[\![\langle \boldsymbol{\tau}\,,\ i\rangle]\!]\ \rho & = & [\texttt{PUSHINT}\ i\ \boldsymbol{\tau}] \\
\mathcal{E}[\![\texttt{let}\ x_1 = e_1\,;\ \ldots\,;\ x_n = e_n\ \texttt{in}\ e]\!]\ \rho & & \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} +\!\!+ \\
 & & \mathcal{E}[\![e]\!]\ \rho' +\!\!+ [\texttt{SLIDE}\ n] \\
 & & \quad \{\text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0]\} \\
\mathcal{E}[\![\texttt{letrec}\ x_1 = e_1\,;\ \ldots\,;\ x_n = e_n\ \texttt{in}\ e]\!]\ \rho & & \\
 & = & [\texttt{ALLOC}\ n] +\!\!+ \\
 & & \mathcal{C}[\![e_1]\!]\ \rho' +\!\!+ [\texttt{UPDATE}\ n-1] +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![e_n]\!]\ \rho' +\!\!+ [\texttt{UPDATE}\ 0] +\!\!+ \\
 & & \mathcal{E}[\![e]\!]\ \rho' +\!\!+ [\texttt{SLIDE}\ n] \\
 & & \quad \{\text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0]\} \\
\mathcal{E}[\![\langle \boldsymbol{\tau}\,,\ e_0 + e_1\rangle]\!]\ \rho & = & \mathcal{E}[\![e_1]\!]\ \rho +\!\!+ \mathcal{E}[\![e_0]\!]\ \rho^{+1} +\!\!+ [\texttt{ADD}\ \boldsymbol{\tau}] \\
 & & \quad \{\text{similarly for arithmetic and comparasion expressions}\} \\
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\ \rho & = & \mathcal{E}[\![e]\!]\ \rho\ +\!\!+ [\texttt{CASEJUMP}\ \mathcal{D}[\![alts]\!]\ \rho] \\
\mathcal{E}[\![\langle \boldsymbol{\tau}\,,\ C_l\ e_1\ \ldots\ e_n\rangle]\!]\ \rho & = & \mathcal{C}[\![e_n]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \mathcal{C}[\![e_1]\!]\ \rho^{+(n-1)} +\!\!+ [\texttt{CONS}\ l\ n\ \boldsymbol{\tau}] \\
\mathcal{E}[\![e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho\ +\!\!+ [\texttt{EVAL}]
\end{array}
$$

Figure 4.8: The $\mathcal{E}$ scheme instrumented for occurrence tags.

$$
\begin{array}{lcl}
\mathcal{C}[\![\langle \boldsymbol{\tau}\,,\ f\rangle]\!]\ \rho & = & [\texttt{PUSHFUN}\ f\ \boldsymbol{\tau}] \\
\mathcal{C}[\![\langle \boldsymbol{\tau}\,,\ x\rangle]\!]\ \rho & = & [\texttt{PUSH}\ (\rho\ x)] \\
\mathcal{C}[\![\langle \boldsymbol{\tau}\,,\ i\rangle]\!]\ \rho & = & [\texttt{PUSHINT}\ i\ \boldsymbol{\tau}] \\
\mathcal{C}[\![\langle \boldsymbol{\tau}\,,\ e_0\ e_1\rangle]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho +\!\!+ \mathcal{C}[\![e_0]\!]\ \rho^{+1} +\!\!+ [\texttt{MKAP}\ \boldsymbol{\tau}] \\
\mathcal{C}[\![\texttt{let}\ x_1 = e_1\,;\ \ldots\,;\ x_n = e_n\ \texttt{in}\ e]\!]\ \rho & & \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} +\!\!+ \\
 & & \mathcal{C}[\![e]\!]\ \rho' +\!\!+ [\texttt{SLIDE}\ n] \\
 & & \quad \{\text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0]\} \\
\mathcal{C}[\![\texttt{letrec}\ x_1 = e_1\,;\ \ldots\,;\ x_n = e_n\ \texttt{in}\ e]\!]\ \rho & & \\
 & = & [\texttt{ALLOC}\ n] +\!\!+ \\
 & & \mathcal{C}[\![e_1]\!]\ \rho' +\!\!+ [\texttt{UPDATE}\ n-1] +\!\!+ \ldots +\!\!+ \\
 & & \mathcal{C}[\![e_n]\!]\ \rho' +\!\!+ [\texttt{UPDATE}\ 0] +\!\!+ \\
 & & \mathcal{C}[\![e]\!]\ \rho' +\!\!+ [\texttt{SLIDE}\ n] \\
 & & \quad \{\text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_n \mapsto 0]\} \\
\mathcal{C}[\![\langle \boldsymbol{\tau}\,,\ C_l\ e_1\ \ldots\ e_n\rangle]\!]\ \rho & = & \mathcal{C}[\![e_n]\!]\ \rho^{+0} +\!\!+ \ldots +\!\!+ \mathcal{C}[\![e_1]\!]\ \rho^{+(n-1)} +\!\!+ [\texttt{CON}\ l\ n\ \boldsymbol{\tau}]
\end{array}
$$

Figure 4.9: The $\mathcal{C}$ scheme instrumented for occurrence tags.

The modified $\mathcal{E}[\![\ ]\!]$ (Figure 4.8) and $\mathcal{C}[\![\ ]\!]$ (Figure 4.9) extract an occurrence tag $\boldsymbol{\tau}$ from the associated Core node in an annotation pair construct and annotate the corresponding instruction with a tag parameter. For further illustration, consider the linear abstract syntax tree representation of the annotated subexpression `(sum 50 60)` from Figure 4.7:

$$\langle\texttt{1, }\langle\texttt{1, }\langle\texttt{1, sum}\rangle\ \langle\texttt{2, 50}\rangle\rangle\ \langle\texttt{3, 60}\rangle\rangle$$

The tagged G-code for this sub-expression is generated by applying the instrumented $\mathcal{E}[\![\ ]\!]$ scheme followed by $\mathcal{C}[\![\ ]\!]$,[1] thus,

$$\begin{aligned}
&\mathcal{E}[\![\langle\texttt{1, }\langle\texttt{1, }\langle\texttt{1, sum}\rangle\ \langle\texttt{2, 50}\rangle\rangle\ \langle\texttt{3, 60}\rangle\rangle]\!]\\
&\quad\Rightarrow \mathcal{C}[\![\langle\texttt{1, }\langle\texttt{1, }\langle\texttt{1, sum}\rangle\ \langle\texttt{2, 50}\rangle\rangle\ \langle\texttt{3, 60}\rangle\rangle]\!] +\!\!+ [\texttt{EVAL}]\\
&\quad\Rightarrow \mathcal{C}[\![\langle\texttt{3, 60}\rangle]\!] +\!\!+ \mathcal{C}[\![\langle\texttt{1, }\langle\texttt{1, sum}\rangle\ \langle\texttt{2, 50}\rangle\rangle]\!] +\!\!+ [\texttt{MKAP 1}] +\!\!+ [\texttt{EVAL}]\\
&\quad\Rightarrow [\texttt{PUSHINT 60 3}] +\!\!+ \mathcal{C}[\![\langle\texttt{2, 50}\rangle]\!] +\!\!+ \mathcal{C}[\![\langle\texttt{1, sum}\rangle]\!] +\!\!+ [\texttt{MKAP 1}] +\!\!+ [\texttt{MKAP 1}, \texttt{EVAL}]\\
&\quad\Rightarrow [\texttt{PUSHINT 60 3}] +\!\!+ [\texttt{PUSHINT 50 2}] +\!\!+ [\texttt{PUSHFUN sum 1}] +\!\!+ [\texttt{MKAP 1}, \texttt{MKAP 1}, \texttt{EVAL}]\\
&\quad\Rightarrow [\texttt{PUSHINT 60 3}, \texttt{PUSHINT 50 2}, \texttt{PUSHFUN sum 1}, \texttt{MKAP 1}, \texttt{MKAP 1}, \texttt{EVAL}]
\end{aligned}$$

When executed, the occurrence tag in an instruction is used to tag the allocated heap cell. For example, the effect of executing the instrumented instructions above is shown in Figure 4.10 omitting the last instruction `EVAL`.

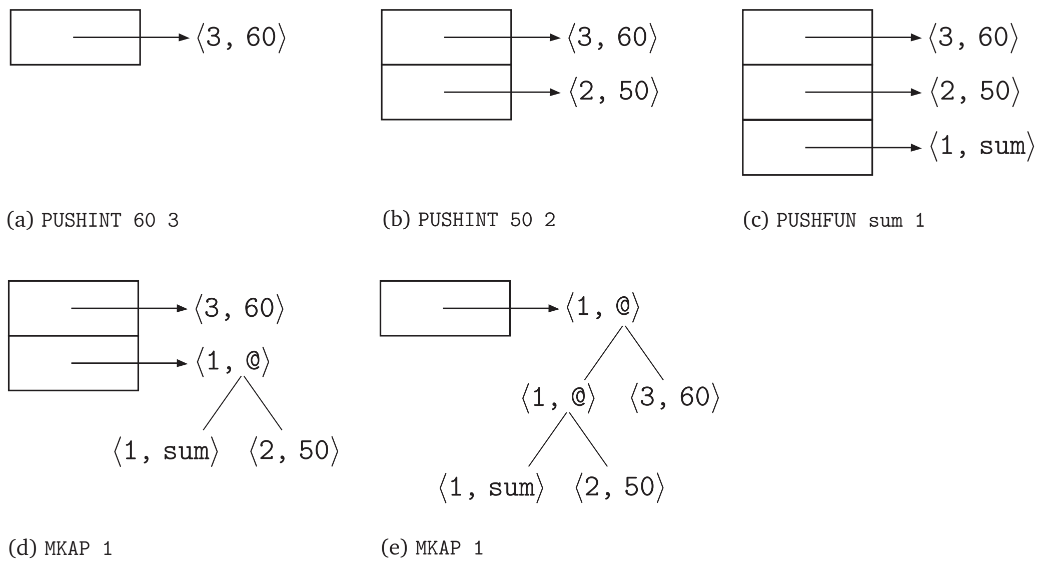


Figure 4.10: Heap cells annotated with occurrence tags.

This way, every heap cell is associated with a corresponding occurrence in the source program through an occurrence tag $\boldsymbol{\tau}$ which is mapped to a profile symbol table allowing *static attributes* of heap cells to be identified when profiling is performed at runtime.

4.2.3 Runtime instrumentation

The runtime system modification is in two parts. The first involves enlarging heap cells to maintain profiling information. The second is a heap profiling sampler

[1] The environment ρ is excluded from the computation to avoid clutter.

which records profiling information from heap cells. These are described respectively.

Heap cells are extended with fields to accommodate both *static* and *dynamic* cell tags. Static tags are used to carry static information about heap cells identified at compile-time whereas dynamic tags are used to carry dynamic information about heap cells identified at runtime.

For static tags, a word is reserved for an *occurrence tag* which points to a record containing profile information in a table. The occurrence tag and the profile information are maintained by the compiler and extended as required by the runtime system (e.g. primitive results information). When a heap cell is allocated, the occurrence tag contained in the instruction responsible for allocation is attached to the reserved field for the static tags. For dynamic tags, a word is reserved for information supplied by the runtime system. By way of example, Figure 4.11 depicts the way a `Cons` cell is tagged with static and dynamic information.

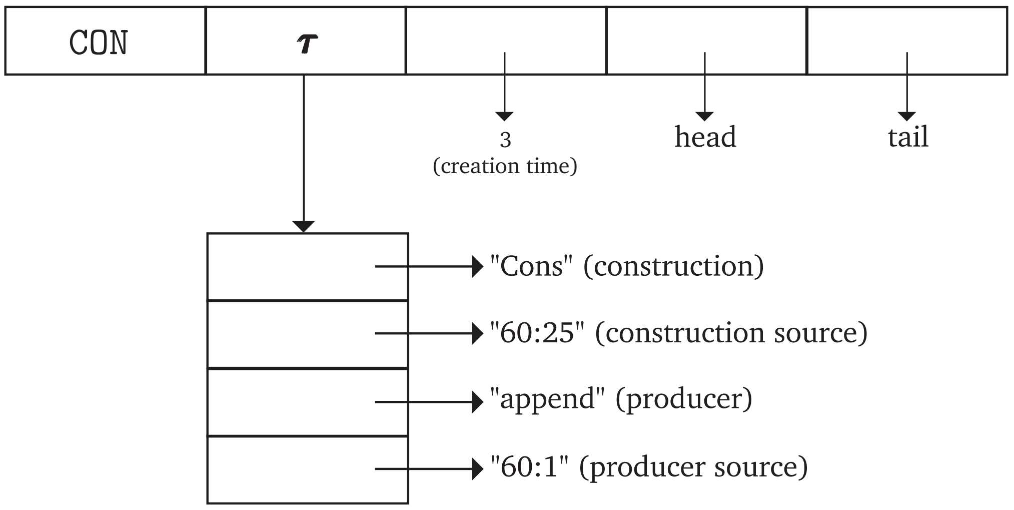


Figure 4.11: Tagged heap cell.

Here, the static tags pointed to by the occurrence tag τ identify static attributes of occurrences using two kinds of information: the occurrence construction that the heap cell represents and its source location; and the producer of the heap cell and its source location. The dynamic tag identifies the creation time of a heap cell. In addition, the runtime system maintains occurrence *allocation counts*: every occurrence is associated with a separate clock which ticks at heap allocation. A heap profile can be obtained by sampling the *live* heap at regular intervals.

One approach is to sample the heap after invoking the garbage collector. While simple in practice, this method requires traversing the heap twice. The cost of heap traversal is reduced by combining the process of heap sampling and garbage collection: while the heap is sampled, a garbage collection is performed [24].

However achieved, care must be taken that the interval-based sampling scheme does not distort memory demands. A common technique is to use real system time as clock-ticks to sample the heap at specific seconds. However, for *space* profiling, a better idea is to sample the heap by fixed *memory allocations*. One consequence of memory allocation-based sampling is that profile results are reproducible and hence portable: i.e. there can be no variations in profiling results if the same program is run twice. Another advantage is the ability to capture previously hidden heap spikes which is a crucial ingredient of a heap profiler, especially for computer systems with increasing speed and for runtime implementations tuned towards fast execution.

Furthermore, for the resulting samples to be consistent, the heap graph ought to be in a proper form before it is traversed. For example, if a graph is sampled when application nodes of a function are not fully constructed, the cost of a function application may vary between different samples. This could hinder the derivation of dynamic attributes of memory (e.g. *lifetime profile*). A solution is to perform a heap sample (or census) when a function is *entered*; that is, just before the ENTER instruction is executed (see Section A.2.2 of Appendix A). This ensures that all the application nodes forming the *spine* of a function are present on the heap (and similarly for variable constructor-field cells). And so sampling is implemented as follows.

A memory meter which ticks at heap allocation is added to the runtime system. The meter advances work with the size of the heap cell in bytes. When code for a function is entered, a routine fielded by memory meter tests whether a sampling interval has elapsed. If so, execution is suspended for a *census* sample of the live heap: the garbage collector is invoked and the entire heap is traversed to collect heap population by counting the number of bytes associated with live cells for each occurrence tag at distinct creation times. The space consumed by cell tags is not counted during a census. When traversal is complete, a census is logged in a file, the meter is reset and execution resumes. By the end of execution, the log file contains a list of censuses along with auxiliary profile information.

The garbage collector has to be modified to accommodate the sampling algorithm. In our implementation, we use the *semispace copying* garbage collection devised by Cheney [72, Chapter 4]. So population counts are collected when heap cells are *evacuated* from the old heap (*from-space*) to the new heap (*to-space*). The modifications to other components of the runtime system are simple but tedious to explain so we move on to some aspects of the profile information produced by the runtime system.

The final profile log recorded by the runtime system consists of a list of censuses in block structure, a profile table and some auxiliary information. Each census is distinguished by a header followed by a series of *census records*. A census header contains the allocation counts when the census was taken. A census record consists of the *occurrence tag*, initially produced by the compiler or else introduced by the runtime system, followed by the associated 1) heap cell population counts in bytes; 2) list of pairs relating creation times with the corresponding memory usage. Creation times are measured by the number of censuses that took place before a heap cell was allocated: heap cells with identical creation time belong to the same *generation* [17]. For example, below is a census block containing three census records.

```
CENSUS 3600
  30 990 1-320 2-340 3-330
  43 792 1-256 2-264 3-272
  44 192 2-192
```

Here, the second census record reports 792 bytes occupied by the heap cells annotated with the occurrence tag 43. This total of 792 bytes is distributed across three creation times: 256 bytes from the first generation, 264 from the second generation and 272 from the third generation.

The profile log provides the necessary information to generate a hotspot profile through post-processing. The occurrence tag is used to distinguish between occurrences of the same symbolic structure and to obtain information from the profile table (e.g. occurrence construction/producer). Because each census record contains generation information, it is possible to derive the *eventual lifetime* of each hotspot by occurrence. By aggregating the population counts of occurrences by construction or producer in every census record, it is also possible to obtain the ordinary *construction*, *producer* or *lifetime* profiles in [1][17] including *restricted* combinations without re-executing the program for profiling.

4.2.4 The ME2G program — Post-processing & graphical display

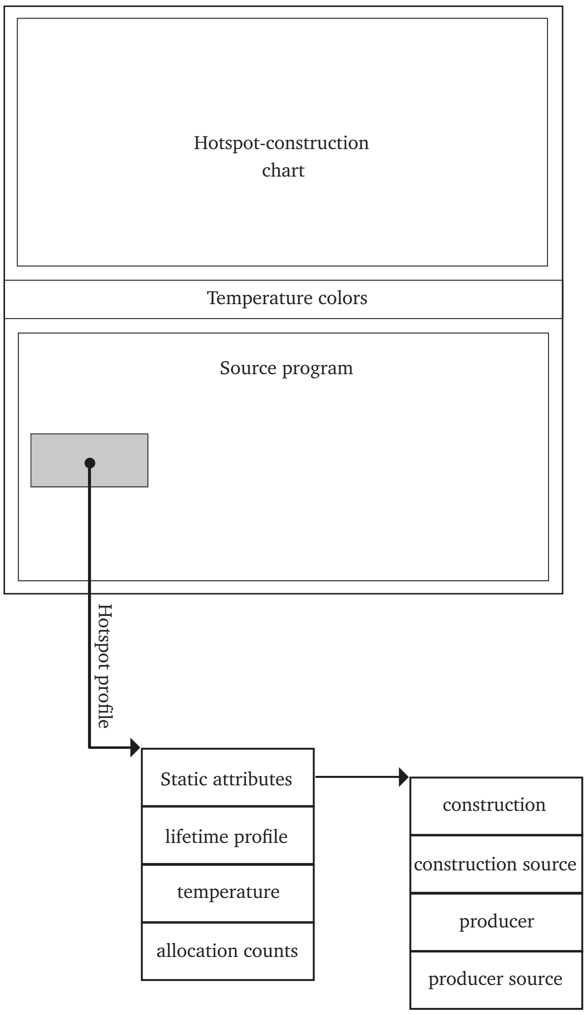


Figure 4.12: A structural prototype of the ME2G program.

The program ME2G *post-processes* the memory profiling information produced by the runtime system and presents it in a graphical display.

Post-processing is in two stages. In the first stage, *hotspots* are derived from *censuses* by calculating the size of occurrence heap graphs and associating each hotspot with a *temperature*, producing a *hotspot-construction* chart. In the second stage, a *lifetime* profile is produced for each hotspot derived from creation times. Post-processing works in the same manner as for HP2PS in [54]. Indeed, this program is used to generate profile charts. Post-processing creation times to derive eventual lifetimes is thoroughly described in [17]; the details are omitted here.

After profiling data are post-processed, they are presented in a graphical display. The structure of this graphical display is composed of three frames as shown in Figure 4.12. The first frame is used to display a hotspot-construction profile loaded from memory. The second frame is preserved for *temperature colours* and the last frame at the bottom is a textual display of the source program with annotated hotspots. Each hotspot is mapped to an individual hotspot profile stored in memory. This profile contains four types of information: *static attributes* which include the producer, construction and their source locations; a lifetime chart; the temperature percentage and allocation counts.

Unlike most previous heap profiling tools, ME2G is interactive. Individual profiles of hotspots are viewed without re-executing the program for profiling.

Additionally, some of the ordinary profiles in [1][17] can be produced, in particular, *producer* and *construction* profiles in addition to their combinations. Appendix B contains the options and commands provided by ME2G.

4.3 Results

In this section we apply hotspot profiling to a series of programs written in the *Pure* language. We conclude with an evaluation of hotspot profiling.

4.3.1 The `maxc` program

We start with the `maxc` program in Figure 4.13, which has some similarities with the `maxw` program in [36]. Given a list of natural numbers, the `maxc` program finds the maximum natural number occurrences and yields as output a list with each maximum occurrence in a singleton list. For a simple example, the maximum occurrences list of `(Cons 6 (Cons 2 (Cons 6 Nil)))` is thus,

```
Cons (Cons 6 Nil) (Cons (Cons 6 Nil) Nil)
```

Overall, list generation is controlled by the function `stream` and supplied to the function `maxc` which returns a maximum occurrences list. A list is produced as follows. The function `mkList` makes a list ranging from `f` to `t` elements, which is then extended with a list of the same size by replicating an element `t`. The function `streams` repeats this process infinitely so the list can be interleaved with replicated elements fairly. The process of generating a list is driven by `stream` forcing a bound `n` on the length of the generated list. The auxiliary functions used by `stream` (hence `streams` and `mkList`) are the usual functions found in a Haskell prelude library (i.e. `replicate`, `append` and `take`). The result of `stream` is supplied to `maxc` which is undefined if a list is empty, otherwise a maximum occurrences list is produced through `maxcc`. This function traverses and compares elements of a list, keeping occurrences of maximum naturals in a separate list which is eventually returned.

```
mkList f t
  | f == t = Cons f (replicate t t)
  | f <  t = Cons f (mkList (f+1) t)

replicate n x = if (n<=0)
                then Nil
                else Cons x (replicate (n-1) x);

append Nil          ys = ys;
append (Cons x xs) ys = Cons x (append xs ys);

take n Nil = Nil;
take n (Cons x xs)
 | n <= 0    = Nil
 | otherwise = Cons x (take (n-1) xs);

stream n f t = take n (streams f t);

streams f t = append (mkList f t) (streams f t);

maxc (Cons n ns) = maxcc Nil 0 (Cons n ns);
maxc Nil         = undefined;

maxcc mos m Nil = mos;
maxcc mos m (Cons n ns)
  | n >  m = maxcc (Cons (Cons n Nil) Nil) n ns
  | n == m = maxcc (Cons (Cons n Nil) mos) m ns
  | n <  m = maxcc mos                     m ns;

main = maxc (stream 220 1 150)
```

Figure 4.13: The `maxc` program.

For the purpose of comparison, we first start by obtaining ordinary profiles (e.g. *construction/producer* in [1]) followed by a series of hotspot profiles. Figure 4.14 shows a producer profile of `maxc`. Heap space is steadily increasing, reaching a peak of approximately 14 kb before execution terminates after about 40 kb of heap allocation. The majority of heap memory is produced by `maxcc` and `take`. So we obtain a restricted profile for the constructions produced by both functions. As Figure 4.15 shows, most of the heap use is associated with the constructions `Cons` and `maxcc`. There is one *occurrence* of `Cons` in the definition of `take`, four `Cons` occurrences and three of `maxcc` on the right-hand side of the function `maxcc`. Are any of these occurrences a *hotspot* associated with high heap demands?

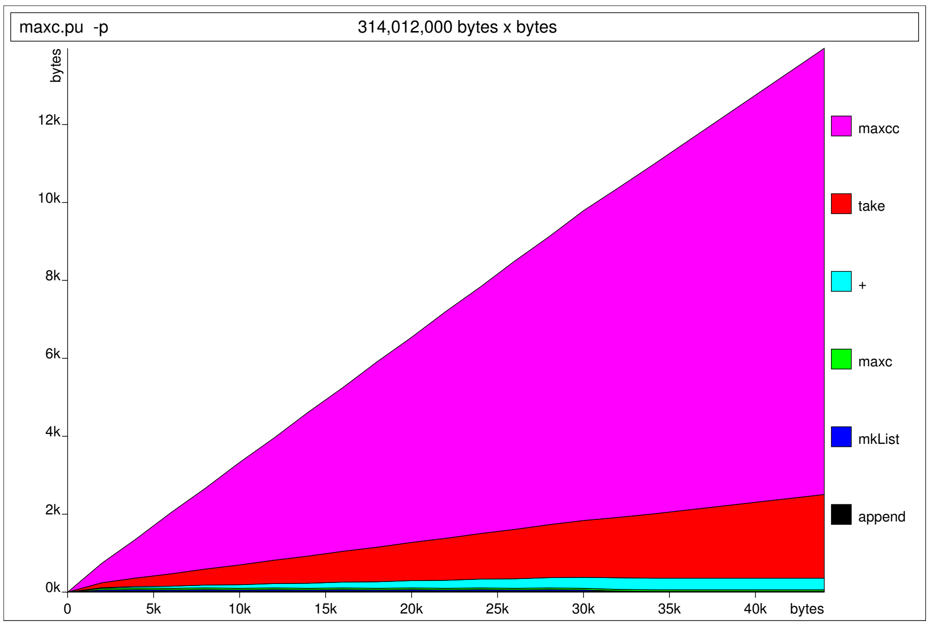


Figure 4.14: A producer profile for `maxc`.

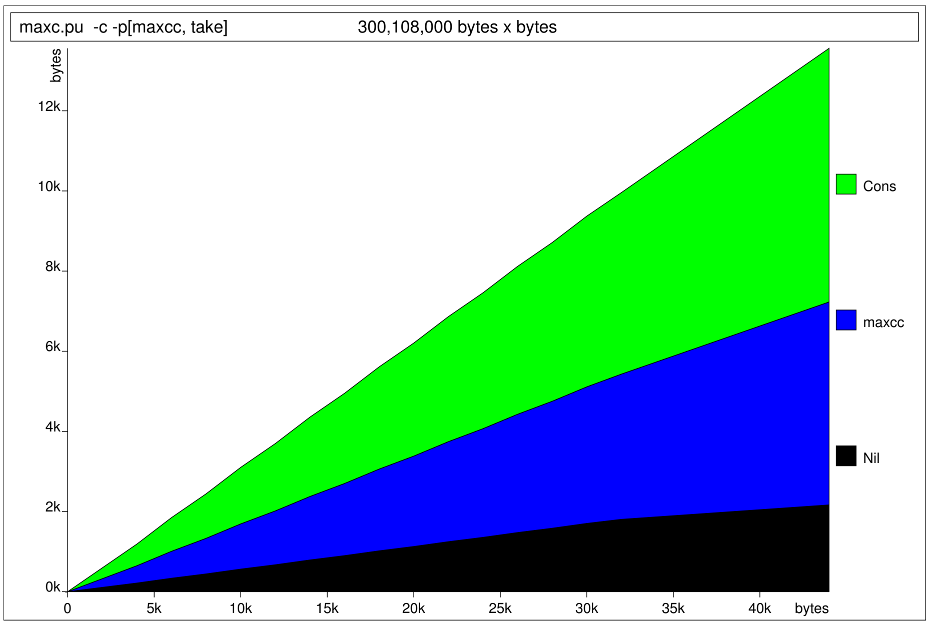
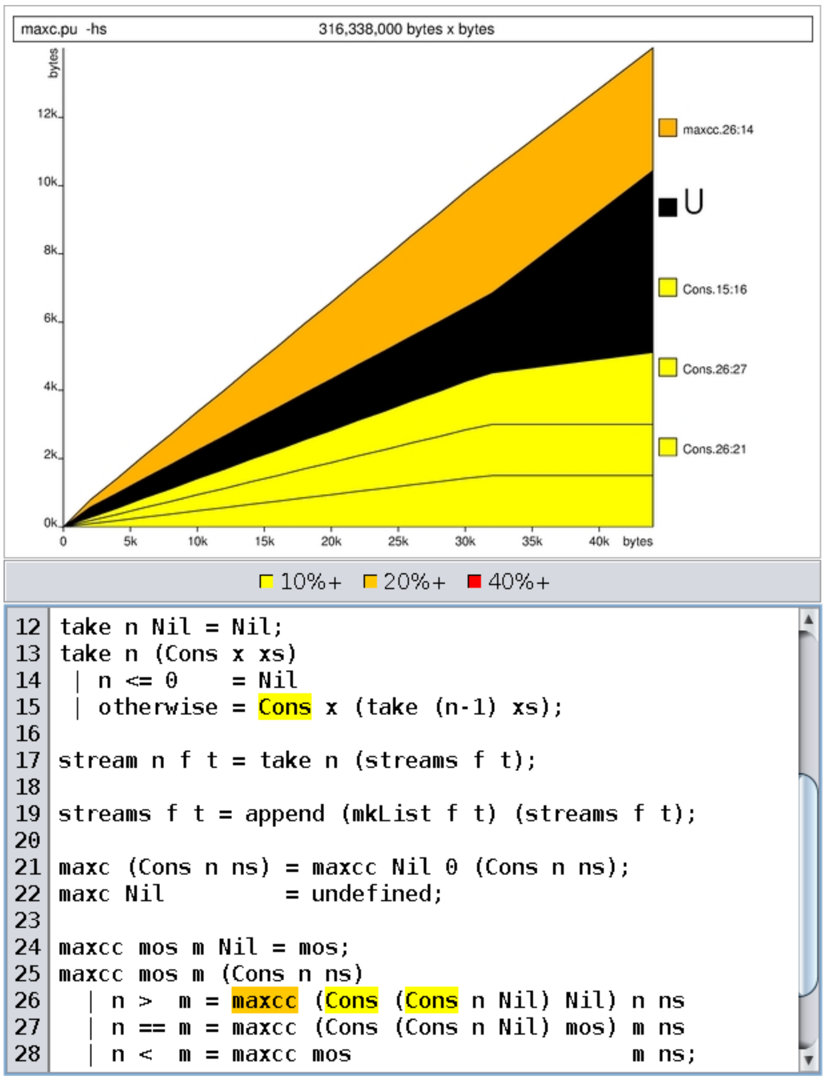


Figure 4.15: A construction profile restricted by the producers `maxcc` and `take`.

The hotspot profile for the `maxc` program in Figure 4.16 shows four hotspots, three of which are most surprising. The first hotspot is `Cons` produced by `take` with a yellow temperature. The other three are `maxcc` belonging to the orange temperature with a heap consumption of 20%+, and two `Cons` occurrences of a yellow temperature occupying 10%+ of heap space. A large amount of heap memory is associated with the last three hotspots constituting a tail-call in the first *guarded equation* of the second equation of `maxcc`. These are of most interest and are examined next. Before attempting to fix the problem concerning the last three hotspots, we first need to study the space behaviour of the corresponding guarded equation intuitively.

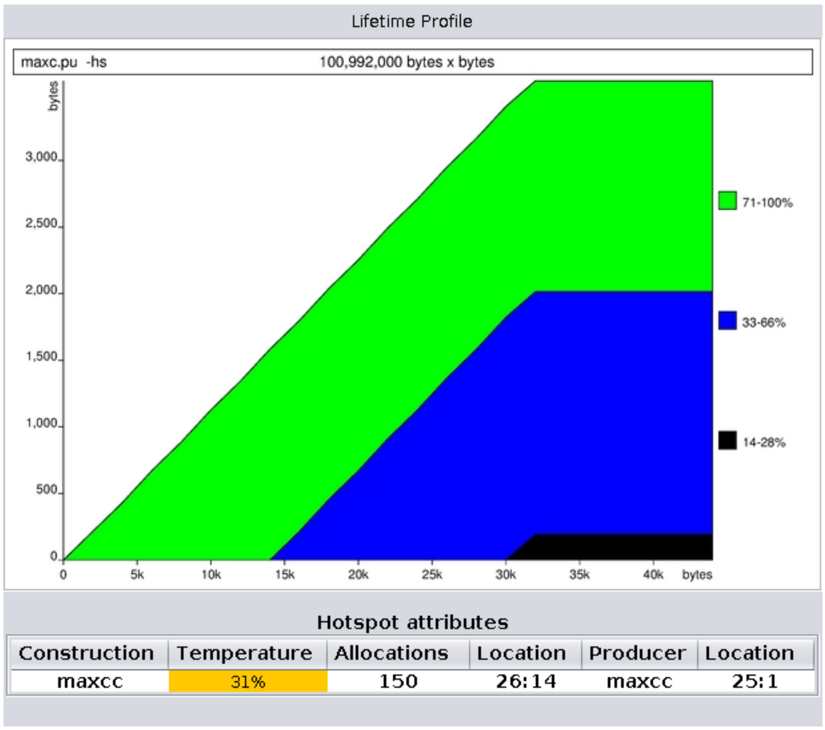


```
12 take n Nil = Nil;
13 take n (Cons x xs)
14   | n <= 0     = Nil
15   | otherwise = Cons x (take (n-1) xs);
16
17 stream n f t = take n (streams f t);
18
19 streams f t = append (mkList f t) (streams f t);
20
21 maxc (Cons n ns) = maxcc Nil 0 (Cons n ns);
22 maxc Nil         = undefined;
23
24 maxcc mos m Nil = mos;
25 maxcc mos m (Cons n ns)
26    | n >  m = maxcc (Cons (Cons n Nil) Nil) n ns
27    | n == m = maxcc (Cons (Cons n Nil) mos) m ns
28    | n <  m = maxcc mos                      m ns;
```

Figure 4.16: A hotspot profile for `maxc`.

In line 26, an element `n` of a list is compared with a current maximum number `m`, an "old" list of maximum-occurrence is replaced with a "new" list containing a list of `n`. So one would expect the two occurrences of `Cons` to occupy minor constant space: two construction cells. However, this is clearly not the case; the hotspot-construction graph (at the top of Figure 4.16) indicates that there are many instances of these `Cons`. Moreover, one can find in the literature assurances that even naive implementations based on *graph reduction* support tail-call optimisation by nature [11]. The `maxcc` hotspot represents a tail-call thus it is expected to occupy constant space at each recursive call, yet the hotspot profile suggests a chain of `maxcc` applications.

Seeking further information, we obtain an individual profile for each of these three hotspots. The individual hotspot profiles give a 13% heap temperature for both `Cons` and 31% for `maxcc` (as illustrated in Figure 4.17). Furthermore, their lifetime profiles indicate that for all three hotspots, heap cells remain in memory for a long period during the execution of the `maxc` program. For example, in Figure 4.17 the majority of `maxcc` heap cells are created early in the computation with a lifetime between 33-66% and 71-100% of the overall program lifetime. This suggests that the heap cells of the hotspots under investigation are *retained*. The function `maxcc` is an obvious retainer since the hotspots in question are only used in the definition of `maxcc`.



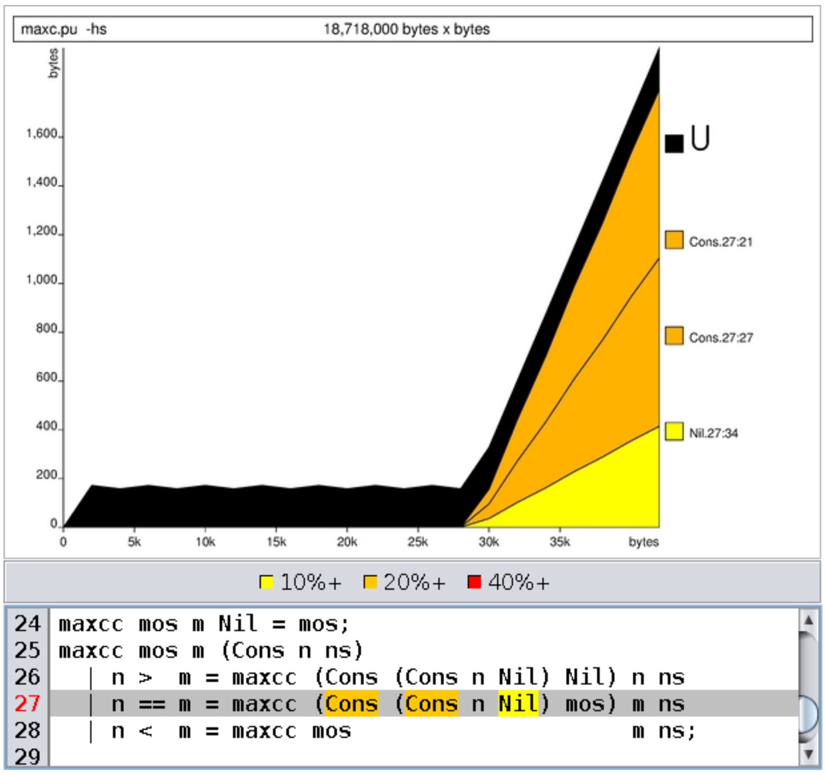
Construction	Temperature	Allocations	Location	Producer	Location
maxcc	31%	150	26:14	maxcc	25:1

Figure 4.17: An individual profile for the hotspot `maxcc`.

After a thorough examination, we concluded that our G-machine implementation suffers from a space leak which causes heap cells to be unnecessarily retained. As mentioned by Johnsson in [73], the problem occurs when tail-calls in non-strict contexts are compiled to *G-code*. In particular, code emitted by the default case of the "optimiser" $\mathcal{E}[\![\,]\!]$ compilation scheme (see Section A.1.3). The resulting code allocates a stack frame for the evaluation of every application to a user-defined function by the EVAL instruction, causing a function to hold a pointer to its arguments on the stack until they are evaluated to *weak head normal form*. The consequence is disastrous space behaviour: *every application to a user-defined function causes the function to retain its arguments until the root of the function application is overwritten with a result*.

In the case of `maxcc` function, tail-recursive calls in every guarded equation allocate a stack frame for evaluating `maxcc` which then retains the supplied list of natural numbers and the maximum occurrence list at every recursive call.[2] Hence the long-lived hotspots `maxcc` and both `Cons` in the hotspot profile of Figure 4.16. This also illustrates the hotspot `Cons` in the definition of `take`, which amongst other functions is retained by `maxcc`.

A solution is support for tail-call optimisation; we added the $\mathcal{S}[\![\,]\!]$ compilation scheme outlined by Johnsson [73] to our compiler using the SQUEEZE and JFUN instructions defined in [11, Chapter 21]. As shown in Figure 4.18, the result is a substantial space improvement: a heap peak decline from 14 kb to 1.8 kb with an *overall cost* decrease of approximately 94%.



```
24  maxcc mos m Nil = mos;
25  maxcc mos m (Cons n ns)
26    | n >  m = maxcc (Cons (Cons n Nil) Nil) n ns
27    | n == m = maxcc (Cons (Cons n Nil) mos) m ns
28    | n <  m = maxcc mos                     m ns;
29
```

Figure 4.18: A hotspot profile for `maxc` after tail-call optimisation.

This new profile shows three hotspots: two `Cons`es associated with a heap temperature of 20%+ and `Nil` belonging to the yellow temperature. These hotspots are associated with the maximum occurrences list as shown in the second guarded equation of `maxcc`. The corresponding heap graph indicates that this list is accumulated to a late stage in the computation, reaching a peak of about 1.7 kb. The (first) `Cons` hotspot shown at the top of the heap profile (below the union band) encourages revisiting the way it is used. A problem is apparent: this outer `Cons` is introduced rather eagerly for every list of a maximum natural, then retained and accumulated until the entire list of maximum occurrences is eventually computed. A solution is to *delay* the creation of these outer `Cons` until the list of maximum occurrences has been calculated, which allows the outer `Cons` to be garbage collected soon after they are created. This desired performance can be accomplished by introducing the auxiliary function below.

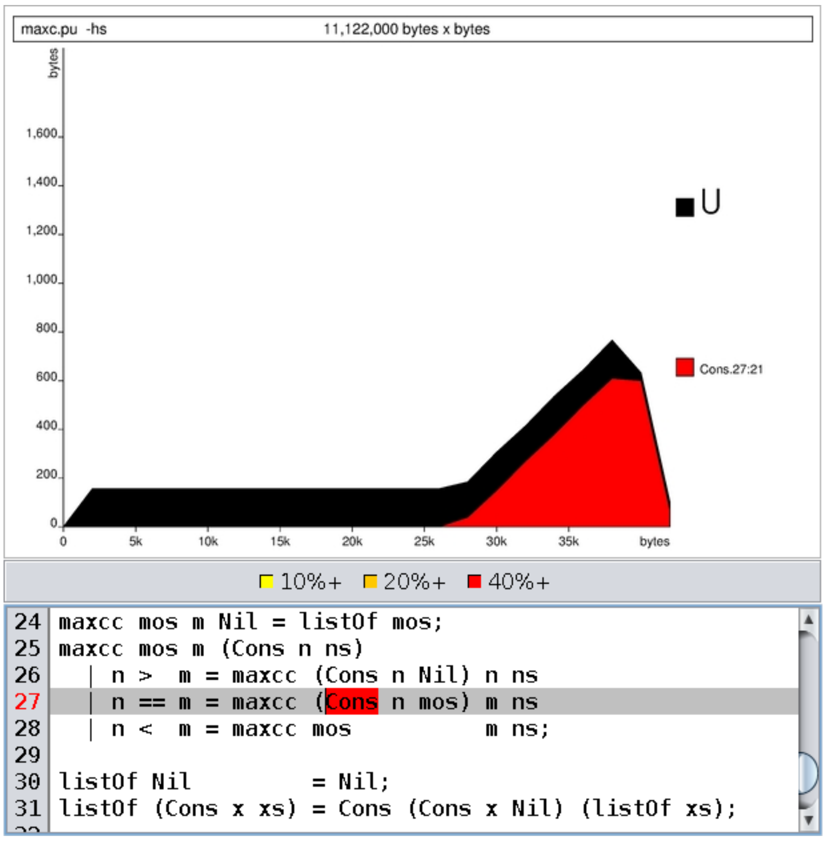
[2]Of course the first equation of `maxcc` also allocates a stack frame for the evaluation of the list argument `mos`, though evaluating a *single-variable* before updating is a special case to preserve laziness [11, Chapter 12].

```
listOf Nil          = Nil;
listOf (Cons x xs) = Cons (Cons x Nil) (listOf xs);
```

That is, the function `listOf` takes a list of x and returns a list with every x in a singleton list. The first equation of the function `maxcc` is therefore modified as follows.

```
maxcc mos m Nil = listOf mos;
maxcc mos m (Cons n ns)
   | n >  m = maxcc (Cons n Nil) n ns
   | n == m = maxcc (Cons n mos) m ns
   | n <  m = maxcc mos          m ns;
```

The new profile applied to the same scale of that in Figure 4.18 is shown in Figure 4.19. The heap peak is significantly reduced (from 1.8 kb to about 0.8 kb) with just one hotspot `Cons` of a red temperature consuming over 40% of the heap. But is this program space-efficient? The accumulation problem for the inner `Cons` of Figure 4.18 remains, so with a large number of maximum occurrences the instances of this `Cons` would increase proportionally to the number of maximum occurrences and the program would eventually run out of space. This prompts a further reformulation of `maxcc` to improve space efficiency.



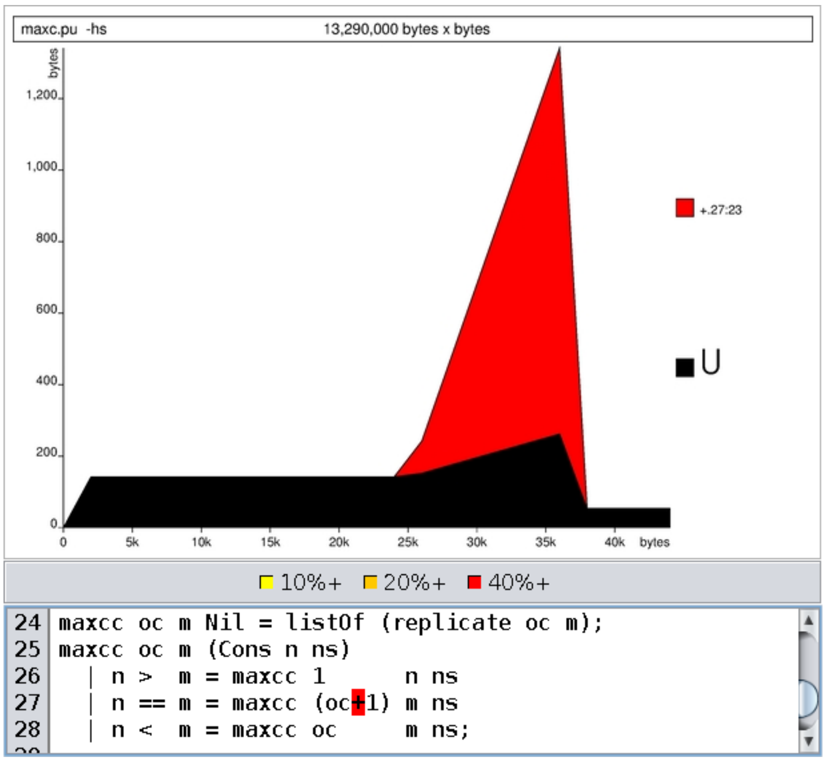
```
24 maxcc mos m Nil = listOf mos;
25 maxcc mos m (Cons n ns)
26    | n >  m = maxcc (Cons n Nil) n ns
27    | n == m = maxcc (Cons n mos) m ns
28    | n <  m = maxcc mos          m ns;
29
30 listOf Nil          = Nil;
31 listOf (Cons x xs) = Cons (Cons x Nil) (listOf xs);
```

Figure 4.19: A hotspot profile for `maxc` after introducing `listOf`.

An obvious alternative to accumulating the maximum occurrences in a list is to count the occurrences of maximum naturals before creating the list of maximum occurrence. One way of achieving this is to replace the list of maximum occurrences `mos` in `maxcc` with a counter `oc`. The counter is set to `1` if a new maximum occurrence is encountered, otherwise incremented. Using this counter and the maximum natural `m`, a maximum occurrence list can be created by replicating `m` for the number of occurrences counter `oc` by the already defined function `replicate`. Accordingly, the function `maxcc` and the first equation of `maxc` are modified as follows.

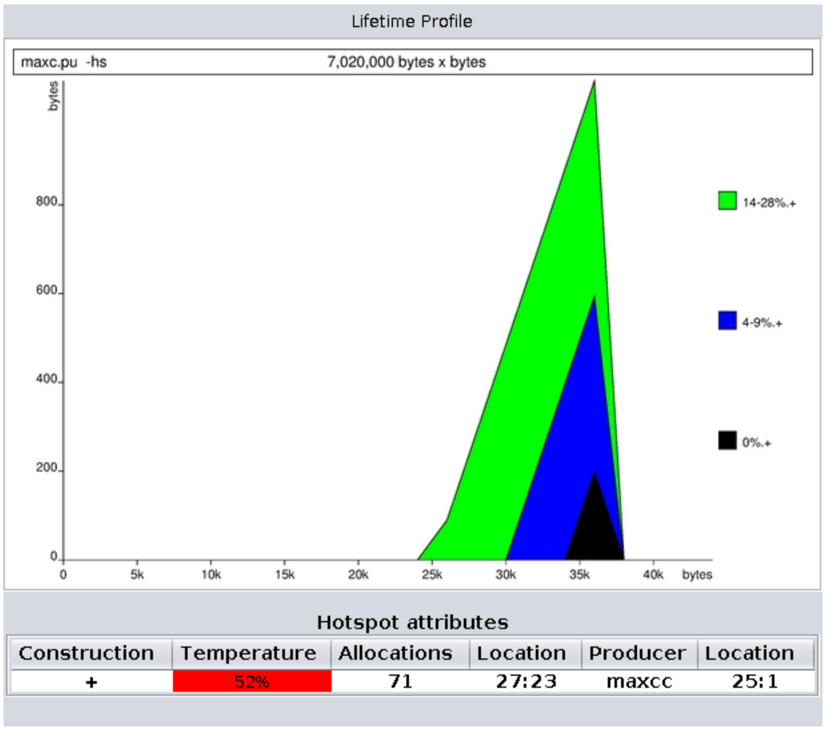
```
maxc (Cons n ns) = maxcc 0 0 (Cons n ns);

maxcc oc m Nil = listOf (replicate oc m);
maxcc oc m (Cons n ns)
  | n >  m = maxcc 1      n ns
  | n == m = maxcc (oc+1) m ns
  | n <  m = maxcc oc     m ns;
```

Unfortunately, the change in the definition of `maxcc` raises heap pressure! Figure 4.20 shows a sharp heap spike of approximately 1.4 kb associated with the hotspot + in the second guarded equation of `maxcc`. Hotspot individual profiles aggregate results (e.g. `Int`) to the primitive that produced them. This is not shown in the key identifiers of the long-lived cell for the hotspot + in Figure 4.21. The problem is apparent: lazy evaluation causes an accumulation of suspended additions to occurrence counts.



```
maxcc oc m Nil = listOf (replicate oc m);
maxcc oc m (Cons n ns)
  | n >  m = maxcc 1      n ns
  | n == m = maxcc (oc+1) m ns
  | n <  m = maxcc oc     m ns;
```

Figure 4.20: A hotspot profile for `maxc` after an occurrences counter `oc` has been introduced.



Hotspot attributes

Construction	Temperature	Allocations	Location	Producer	Location
+	52%	71	27:23	maxcc	25:1

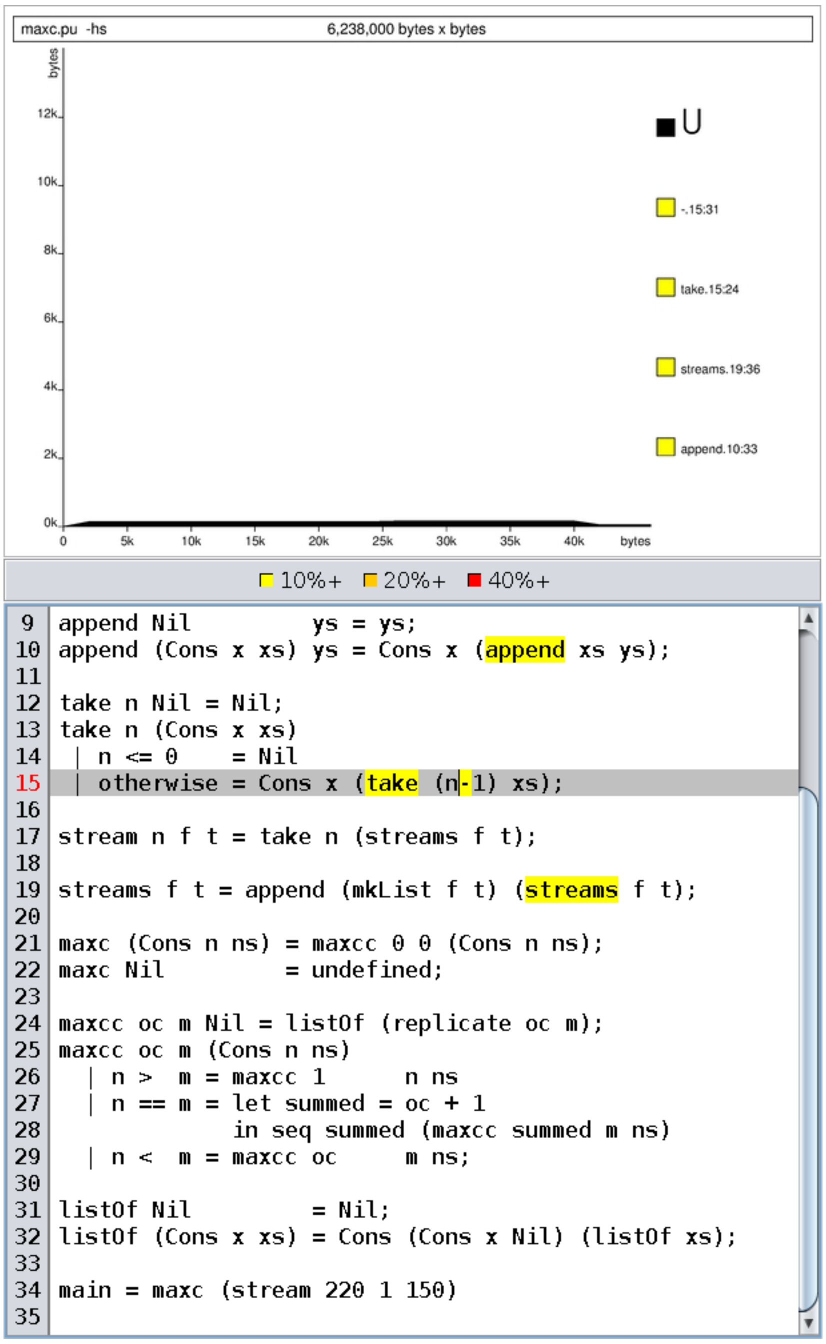
Figure 4.21: An individual hotspot profile for + after introduction of an occurrences counter oc.

One way to proceed is to force the additions using the built-in function seq [44]. This can be accomplished by changing the right-hand side of the second guard in maxcc to the following.

```
let summed = oc + 1
in seq summed (maxcc summed m ns)
```

Here, seq evaluates its first argument (i.e. the occurrence addition oc + 1) and returns its second argument. The use of a *let*-construct allows the evaluated summation (summed) to be shared by maxcc. The result is shown in the final hotspot profile of Figure 4.22 applied to the same scale of the first profile of the maxc program in Figure 4.16.

The space cost of the final maxc program is significantly improved, decreasing the heap peak from 14 kb to about 0.2 kb and reducing the overall space cost by approximately 98%. The maxc program finally enjoys a constant space execution.



```
append Nil          ys = ys;
append (Cons x xs) ys = Cons x (append xs ys);

take n Nil = Nil;
take n (Cons x xs)
  | n <= 0    = Nil
  | otherwise = Cons x (take (n-1) xs);

stream n f t = take n (streams f t);

streams f t = append (mkList f t) (streams f t);

maxc (Cons n ns) = maxcc 0 0 (Cons n ns);
maxc Nil         = undefined;

maxcc oc m Nil = listOf (replicate oc m);
maxcc oc m (Cons n ns)
   | n >  m = maxcc 1      n ns
   | n == m = let summed = oc + 1
              in seq summed (maxcc summed m ns)
   | n <  m = maxcc oc     m ns;

listOf Nil         = Nil;
listOf (Cons x xs) = Cons (Cons x Nil) (listOf xs);

main = maxc (stream 220 1 150)

```

Figure 4.22: The final hotspot profile of `maxc` after introducing `seq`.

4.3.2 The `execute` program

The next example is the `execute` program defined in Figure 4.23. The `execute` program implements a stack-based *abstract machine* for evaluating arithmetic expressions built from integers[3]. The abstract machine operates on a set of instructions and produces an integer value. These instructions are given by the algebraic data type `Instruction`:

```
data Instruction = PUSH Int
                 | ADD
                 | MUL
                 | SUB
                 | DIV;
```

The meaning of these instructions is explained shortly. Arithmetic expressions are built from *code* which comprises a list of `Instructions`. For example, code for the expression $5 + 10$ is represented as follows.

```
Cons (PUSH 10) (Cons (PUSH 5) (Cons ADD Nil))
```

Such code is executed using an *evaluation stack*. A stack in this context is merely a list of integers and code operates on the stack by a list of push and arithmetic instructions. The meaning of the code is defined by the function `exec` below. When supplied with an empty initial stack, code is executed and the result is found on top of the stack (or actually, as the head of a list).

```
exec Nil                s = s;
exec (Cons (PUSH n) c) s = exec c (Cons n s);
exec (Cons ADD c)       s = exec c (Cons (add s) (pop 2 s));
. . .
```

Here, the PUSH instruction pushes an integer value `n` onto the top of the stack. The ADD instruction pushes the sum of the top two elements onto the stack and pops the summed elements off the stack using the auxiliary functions `add` and `pop`, respectively; MUL (multiply), SUB (subtract) and DIV (divide) operate in a similar way.

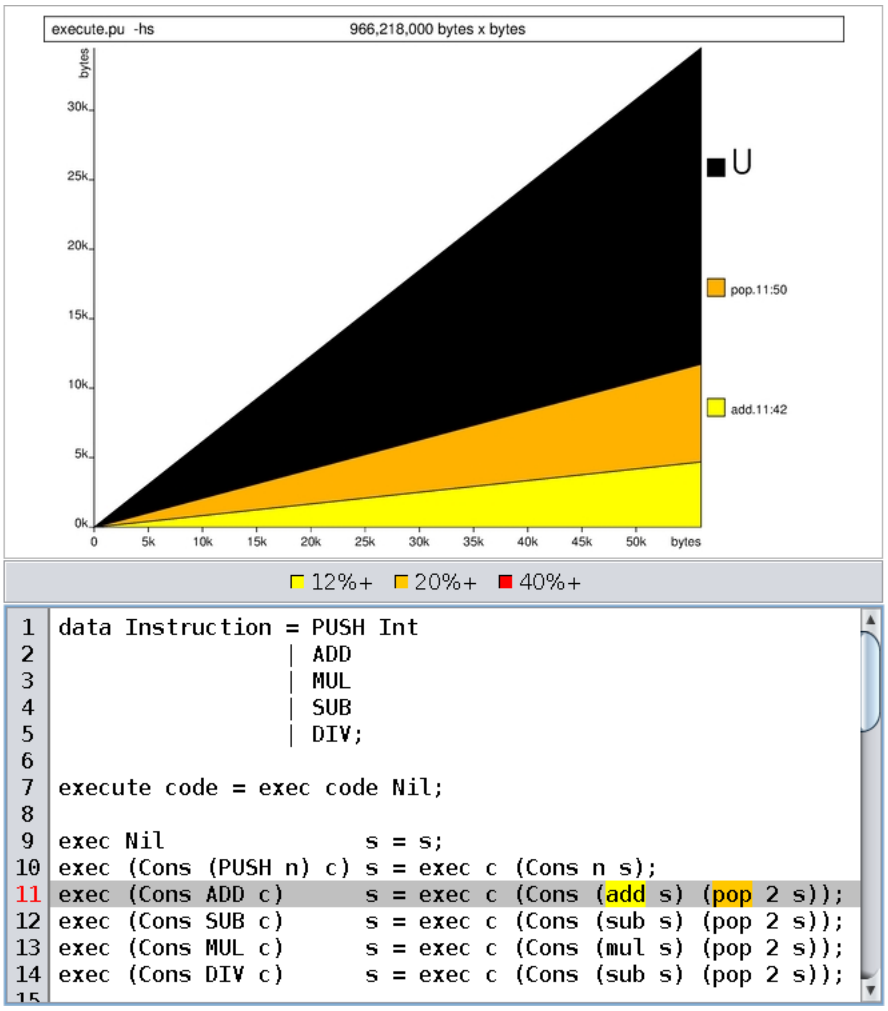
For this illustration, a function `makeCode` is used to generate code of additions ranging from the arguments `f` to `t`. Finally, code produced by `makeCode` is supplied to the function `value` which invokes the abstract machine executor (`execute`) and extracts an integer result from the top of the evaluation stack.

A hotspot profile for the `execute` program is shown in Figure 4.24. Heap space steadily increases, approaching a peak of about 35 kb by the end of computations after approximately 55 kb of heap memory allocations. A large section of the heap (associated with the union band U) seems to be evenly distributed. There are just two hotspots residing in the third equation of the function `exec`: `add` belonging to the yellow temperature and `pop` with a temperature of 20%+. The individual profiles (e.g. in Figure 4.25) indicate that the majority of accumulated heap cells are longed-lived (e.g. in the 55-100% band). With this information and examining the relevant equation, a problem is apparent. Lazy evaluation causes an accumulation of suspended `add` and `pop` computations in the evaluation stack: the elements of the stack are added, then popped off the stack after the entire accumulation while retaining the stack in the heap.

[3]This program is based on the abstract machine in [2, Chapter 31].

```
data Instruction = PUSH Int
                 | ADD
                 | MUL
                 | SUB
                 | DIV;

execute code = exec code Nil;

exec Nil               s = s;
exec (Cons (PUSH n) c) s = exec c (Cons n s);
exec (Cons ADD c)      s = exec c (Cons (add s) (pop 2 s));
exec (Cons SUB c)      s = exec c (Cons (sub s) (pop 2 s));
exec (Cons MUL c)      s = exec c (Cons (mul s) (pop 2 s));
exec (Cons DIV c)      s = exec c (Cons (sub s) (pop 2 s));

add Nil                    = undefined;
add (Cons x Nil)           = undefined;
add (Cons m (Cons n s))    = n + m;

mul Nil                    = undefined;
mul (Cons x Nil)           = undefined;
mul (Cons m (Cons n s))    = n * m;

sub Nil                    = undefined;
sub (Cons x Nil)           = undefined;
sub (Cons m (Cons n s))    = n - m;

div Nil                    = undefined;
div (Cons x Nil)           = undefined;
div (Cons m (Cons n s))    = n / m;

pop n Nil = Nil;
pop n (Cons x xs)
  | n == 0    = Cons x xs
  | otherwise = pop (n-1) xs;

makeC f t = if (f > t)
            then Nil
            else Cons (PUSH f) (Cons ADD (makeC (f+1) t));

makeCode f t = Cons (PUSH f) (makeC (f+1) t);

value code = case (execute code) of {
               Cons x xs -> x;
               Nil       -> undefined
             };

main = value (makeCode 1 400)
```

Figure 4.23: The execute program.



```
data Instruction = PUSH Int
                 | ADD
                 | MUL
                 | SUB
                 | DIV;

execute code = exec code Nil;

exec Nil               s = s;
exec (Cons (PUSH n) c) s = exec c (Cons n s);
exec (Cons ADD c)      s = exec c (Cons (add s) (pop 2 s));
exec (Cons SUB c)      s = exec c (Cons (sub s) (pop 2 s));
exec (Cons MUL c)      s = exec c (Cons (mul s) (pop 2 s));
exec (Cons DIV c)      s = exec c (Cons (sub s) (pop 2 s));
```

Figure 4.24: A hotspot profile of `execute` program.

Instead of delaying the addition and removal of stack elements, it is far more efficient to replace the top two elements of the stack by their sum through pattern matching in the definition of the ADD instruction. This replacement can be achieved by modifying the third equation of `exec` as follows:

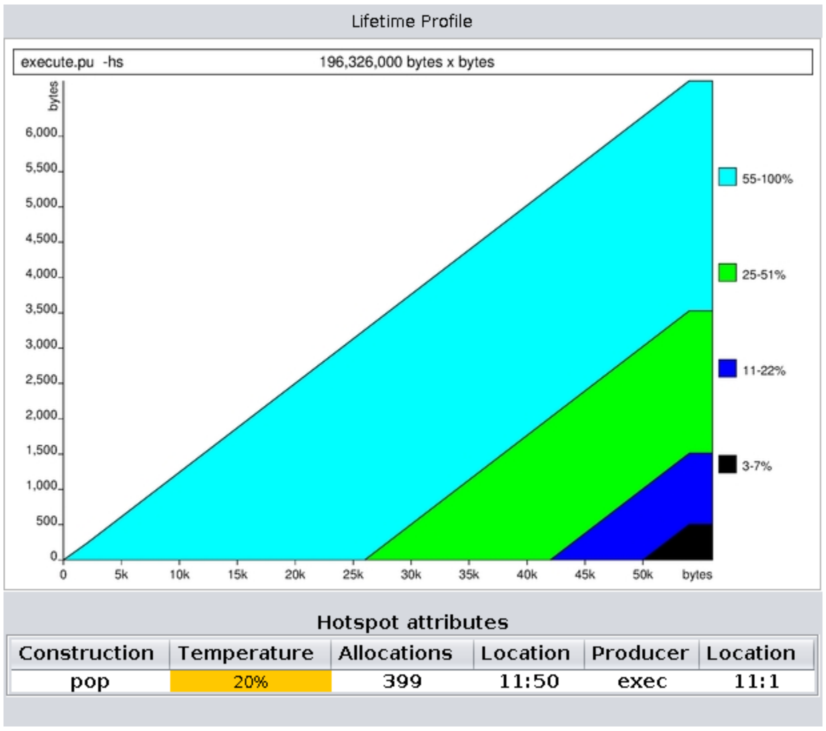
```
exec (Cons ADD c) (Cons f (Cons s ss))
    = exec c (Cons (s+f) ss);
```

However, we are still faced with accumulated suspensions of summation proportional to the occurrences of ADD in the code. One remedy is to force addition using `seq` as shown below (in a similar manner to the previous section).

```
exec (Cons ADD c) (Cons f (Cons s ss))
    = let added = s + f
      in seq added (exec c (Cons added ss));
```

The effect of these modifications is a reduction to about half of the memory demands.

The new profile in Figure 4.26 reveals four hotspots comprising the list of instructions. From the hotspot-construction graph, it seems that the list of instructions is retained. This is surprising as code is generated by need and discarded in the process of execution — by `makeC` and `exec`, respectively. This leaves us with what triggers the execution machinery:



Hotspot attributes

Construction	Temperature	Allocations	Location	Producer	Location
pop	20%	399	11:50	exec	11:1

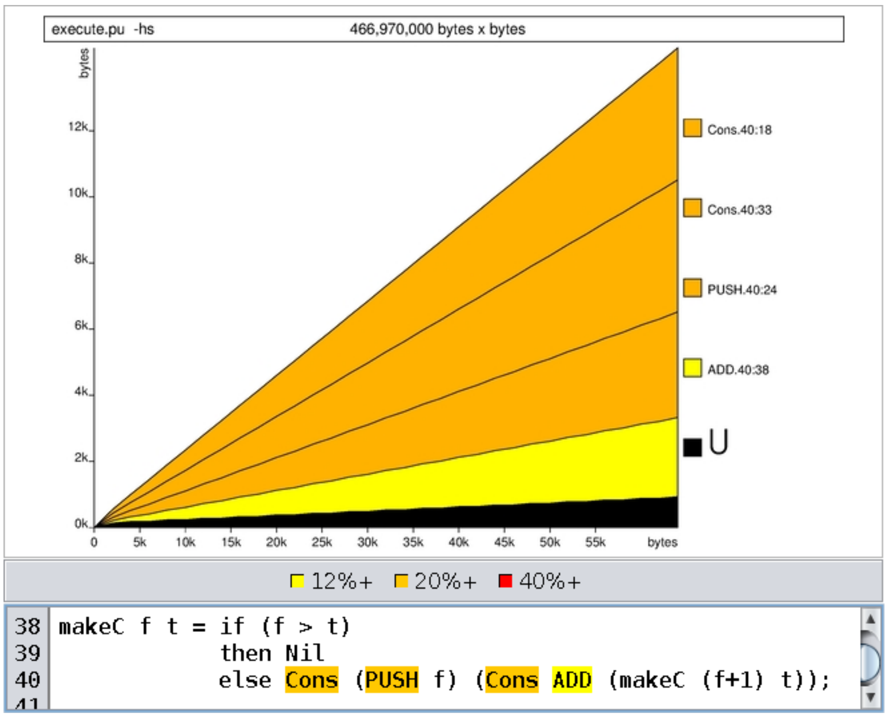
Figure 4.25: An individual hotspot profile of the pop hotspot.

```
value code = case (execute code) of {
               Cons x xs -> x;
               Nil       -> undefined
             };
```

Here, the intention is to execute code (a list of instructions) and extract a value from the head of the resulting list. Unfortunately, this rather innocent looking code leads to a space leak connected with the implementation of case expressions: the argument `code` cannot be released until the application `execute code` has been evaluated. As a consequence, the entire list of instructions is unnecessarily retained until a result is returned, which is just before the program completes its execution. This problem can be solved by *stack stubbing* (e.g. see Section A.6 in [61]) which in this instance would overwrite the stack pointer to the argument `code` with a `STUB` node, thereby preventing the list of instructions from being retained. However, there is also a simpler fix at source level: we merge code generation and extraction, as shown below.

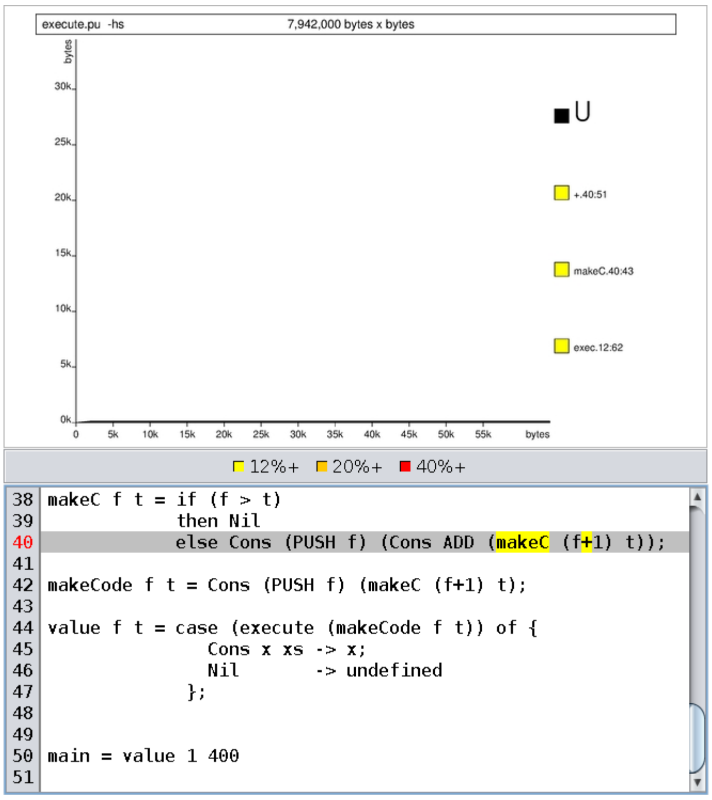
```
value f t = case (execute (makeCode f t)) of {
               Cons x xs -> x;
               Nil       -> undefined
             };
```

The hotspot profile after the modifications above is shown in Figure 4.27. Compared to the overall cost of the profile in Figure 4.24, the gain is a heap reduction of about 99%. The final version of this abstract machine executes code in constant space.



```
38 makeC f t = if (f > t)
39             then Nil
40             else Cons (PUSH f) (Cons ADD (makeC (f+1) t));
```

Figure 4.26: A hotspot profile of `execute` program after strictifying the stack.



```
38 makeC f t = if (f > t)
39             then Nil
40             else Cons (PUSH f) (Cons ADD (makeC (f+1) t));
41
42 makeCode f t = Cons (PUSH f) (makeC (f+1) t);
43
44 value f t = case (execute (makeCode f t)) of {
45               Cons x xs -> x;
46               Nil       -> undefined
47             };
48
49
50 main = value 1 400
51
```

Figure 4.27: A hotspot profile of `execute` program after removing the `code` argument from `value`.

4.3.3 Evaluation

So what have we achieved? Using the hotspot profile allowed us to *locate hotspots* and therefore reduce the *overall cost* of the `maxc` program (in Section 4.3.1) and the `execute` program (in Section 4.3.2) by 98% and 99%, respectively.

Over the original *construction*, *producer* and *lifetime* heap profiles (e.g. in [1] [17][54]), hotspot profiling provides two notable advantages. Furnishing a heap *section* with several attributes can be more productive than iteratively applying a profile of a different kind to gather information about memory, particularly *producer* and *construction* profiles since they are usually first resort. Most importantly, the ability to pinpoint a hotspot by occurrence rather than a reference to an entire body of a function can give an insight into the space behaviour of parts of expressions with the aid of additional information about hotspots.

The individual hotspot profiles are helpful mostly to investigate memory characteristics of hotspots. For most of the hotspots that we have encountered, their lifetime profiles indicate that heap cells are long-lived. Our immediate question was "who retains these hotspots?" To answer this question, we had to use our knowledge of lazy evaluation and the underlying implementation. A retainer-by-occurrences profile would have therefore enhanced the profiling experience by several factors.

Classifying hotspots by predefined temperatures highlights those associated with the highest amount of heap contribution. In general, it is easier to deal with space leaks caused by a few hotspots with a focus on the most space-demanding ones. For many programs, the default temperatures (i.e. `10%+`, `20%+` and `40%+`) provide a good heap distribution for hotspots. However, this is not always the case. Using the default temperatures for the `execute` program in Section 4.3.2, we encountered six hotspots; raising the minimum (yellow) temperature by 2% ruled out four hotspots which allowed us to focus on the two hotspots associated with the highest percentage of heap memory. Ignoring occurrences occupying less than 10% is remarkably effective. It rules out occurrences with even distribution of a small heap contribution in a profile, which then focuses on hotspots.

The effectiveness of hotspot profiling greatly depends on the program at hand. Not all programs have hotspots. However, many of the programs we have profiled do have one or more hotspots. Often, space occupied by these hotspots can be reduced by a simple reformulation of the source program. This was the case in Section 4.3.1 and 4.3.2. The hotspot profiler is of no help for programs that have no hotspots. These programs tend to have even distributions of heap memory. An example of this is the tautology checker program in [2, Chapter 10]. As shown in the hotspot-construction profile of Figure 4.28, there are no hotspots. Heap space is evenly distributed. The corresponding ordinary construction profile shown in Figure 4.29 is more informative. This program seems well-behaved.

In many ways, our heap profiling implementation resembles the heap profiling system in [1][17] and it suffers from the same problem: there is no direct means of differentiating between many instances of the same function application unless otherwise renamed.

So far we have illustrated the use of the hotspot profiler on small and limited programs. In Chapter 6, we demonstrate hotspot profiling for a larger and more complex program with further evaluation.

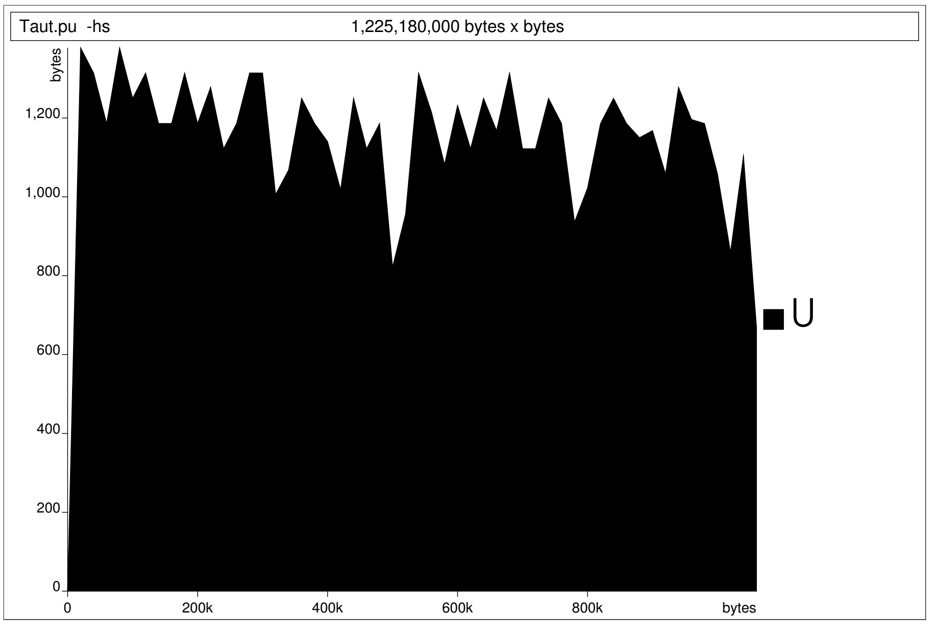


Figure 4.28: A hotspot-construction profile for the Taut program.

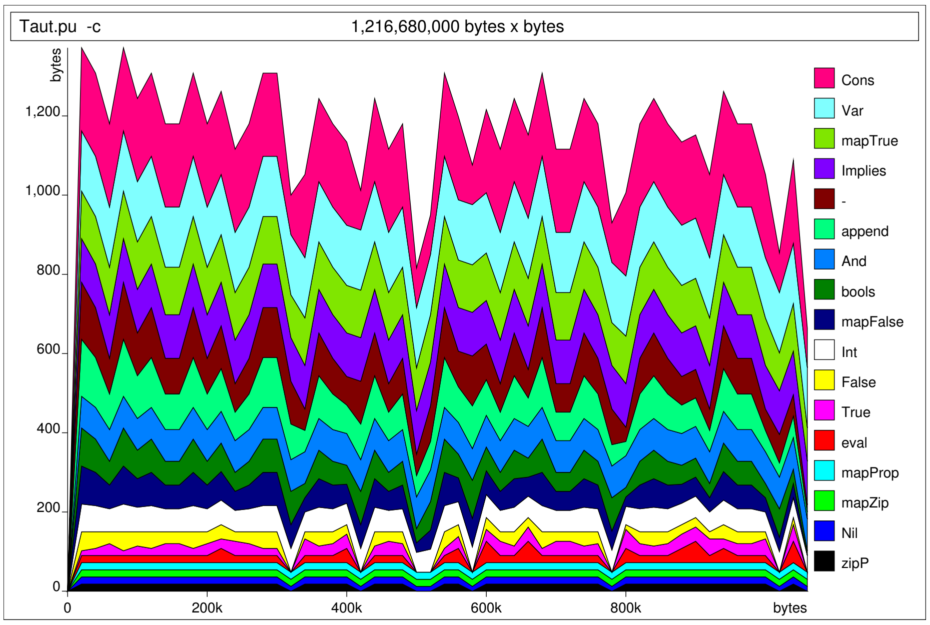


Figure 4.29: A construction profile for the Taut program.

Chapter 5

A Stack Profiler

ïż£In this chapter, we introduce a stack profiler which provides summary charts for *producers* of stack frames and the *constructions* they represent. Section 5.1 explains the motivation for stack profiling. Section 5.2 presents the design of the stack profiler for producers and constructions. Section 5.3 presents a scheme for attaching profiling information to the stack. Section 5.4 describes the implementation of the stack profiler for a lazy functional language. Section 5.5 concludes with illustrative applications of stack profiling to programs.

5.1 Motivation

For the last two decades, various lazy functional implementations have offered the programmer *heap profiles* which yield valuable information about memory. However, space leaks causing *stack overflows* are common in functional programming, partly as a result of the recursive nature of function definitions. Programmers are pressing for tools to diagnose stack space memory [74]. So we investigate the design and implementation of a stack profiler which yields information about memory in a similar manner to heap profiles.

5.2 Stack Profiling

Recall from Section 3.3 that the abstract G-machine is a stack-based implementation executing lazy functional programs by way of normal-order graph reduction. The stack is used to maintain the evaluation of function applications by storing pointers to function arguments and to the sub-expressions of instantiated bodies of functions in a *dump* of stacks.

In practice, graph reduction is implemented by way of *stack frames*[1]. Stack content is a series of frames for dynamically nested function applications. Stack frames are dynamically allocated (pushed) and deallocated (popped) from the stack with function calls, in a last-in-first-out manner.

When a program is executed, the stack memory dynamically grows and shrinks with the chain of nested function applications. As a function call is made, a new frame is stored in stack memory, as the function returns, the frame memory is

[1]Also called *activation records, call stack* or *runtime stack.*

restored for later re-use. When the supply of stack memory runs out, the implementation terminates execution, raising a *stack overflow*.

In current implementations of functional languages, stack overflows are notorious. The programmer writes elegant recursive definitions but the implementation may impose excessive stack space demands. When a stack overflow strikes, the programmer might raise several questions analogous to that of heap memory [1], for example, "Which functions occupy most of the stack frames?", "Who introduced those frames into the stack?" We provide a stack profiler which supplies the programmer with answers to these questions.

The stack profiler is implemented in two components: an *instrumented compiler* which generates profiling information and a *post-processor* which transforms profiling information to graphical form.

The instrumented compiler *marks* every stack frame with two tags. These tags identify:

- The (immediate) function that *produced* the frame in the stack.
- The *construction* that the stack frame represents.

For producers, we take the names of top-level functions. The initial stack frame is reserved for the SYSTEM and tagged accordingly. For constructions, we take the name of the function component of a *closure*. Every stack frame is assumed to represent a (yet to be evaluated) closure. We say more about marking frames with constructions and producers in Section 5.3.

When a program is executed for profiling, the profiler suspends execution at specified intervals for a stack *census*: the implementation traverses the runtime stack, collecting population counts and information from each frame. After stack traversal is complete, the gathered information is logged in a file and the execution of the program is resumed. When execution terminates, the log file includes a profile of stack frames at each interval.

A post-processor transforms a stack profile to a graphical chart. An example stack chart is shown in Figure 5.1. The overall shape of the graph shows how the amount of stack storage (vertical axis) varies over the *stack allocation counts* (horizontal axis) that the program takes to execute. Stack storage and allocation counts are measured in bytes. In this example, the stack reaches a peak of about 400 bytes over approximately 6 kb of stack allocations. The profiling overheads are excluded from the chart.

The title in the top area of the stack profile is in three parts: the name of the profiled program followed by profiling options, a measure of the *overall stack cost* of the program in bytes (stack storage) $\times$ bytes (stack allocation counts).

Bands in the graph show the amount of storage occupied by stack frames tagged with each of the identifiers in the keys to the right of the graph. The graph in Figure 5.1 is a construction profile, so each key identifies the top-level function whose applications the stack frames represent.

The graphical presentation of stack profiles is similar to the original heap profiles in [1]. Indeed, we use the heap profile post-processor (in Appendix B) to generate stack charts.

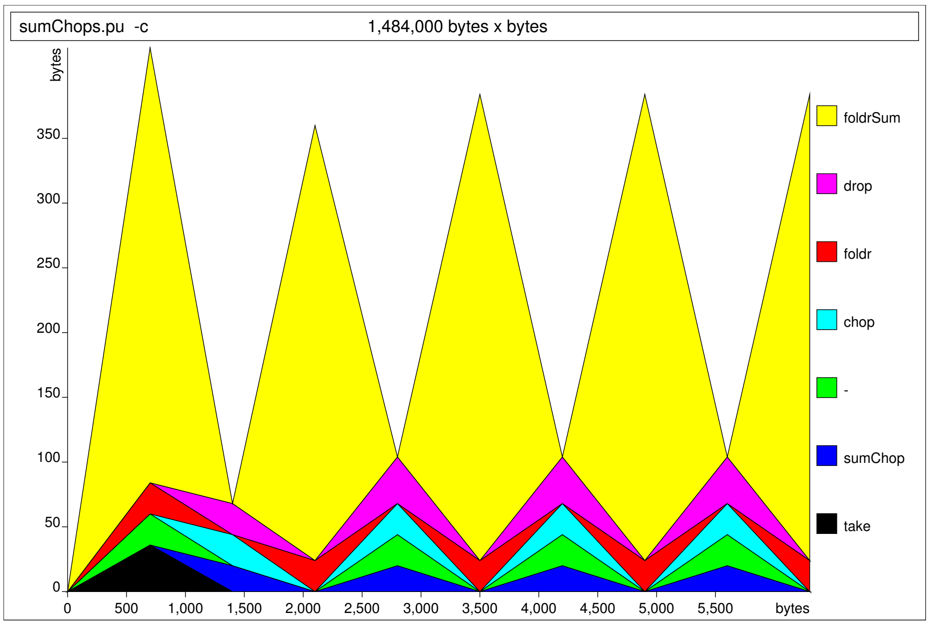


Figure 5.1: A stack profile example.

5.3 Tagging the Stack

In this section we describe a scheme for tagging stack frames with constructions and producers, focusing on construction of frames. We first start by describing execution machinery in relation to the runtime stack.

In our implementation of the G-machine, the stack frame configuration of an executing function is shown in Figure 5.2 [11]. The stack grows downwards and the frame in discussion is on top of the runtime stack. The section at the bottom of the frame includes bookkeeping information associated with each frame to maintain the runtime stack, in particular the *return address* which points to the code sequence of the caller, to be executed when the current function returns, and the *old frame pointer* which links stack frames together. The top frame is associated with two registers: the *frame pointer* (fp) which points to the old frame pointer and the *stack pointer* (sp) which points to the top of the stack. This implementation technique is reminiscent of stack frames of imperative languages. The section on top of the bookkeeping information, starting from the pointer to the root of the current *redex* (or current function), the arguments to the function and the intermediate values of the function, we term the *frame context*.

Therefore, during the execution of a function, its arguments are on top of the stack and beneath them remain a pointer to the root of the function application to be updated with the result of the function application. The execution of a function in relation to a stack frame is performed as follows.

When a call to a nested function is made a new stack frame is allocated on top of the current stack, pushing bookkeeping information on top of the frame with the return address as part of the call. The root of the function application is pushed onto the top of the stack and the machine enters the UNWIND state; the spine of the function application graph is unwound, pushing pointers to the (unevaluated) arguments of the function along the way on the stack. Subsequently, the execution of the current function begins, pushing pointers to its required arguments, and to the sub-expressions of the current function body as it is built on top of the stack. Meanwhile, if a nested function within the body of the current function is called, a new stack frame is allocated for the evaluation of the nested function. When execution of the current function completes, the root of the function application graph is updated with a result, a jump is made to the return address and the current stack frame is popped off the stack, leaving a pointer to the updated result on top of the restored frame. The caller continues its execution and this cycle is repeated for function applications.

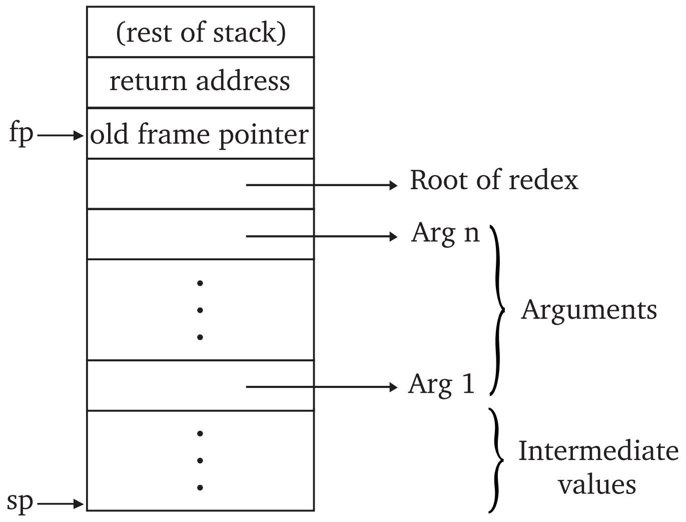


Figure 5.2: The stack configuration of an executing function. Some components of the stack frame are excluded to simplify discussion.

The construction of a frame can be regarded as the name of the executing function. The producer of a frame can be regarded as the name of the top-level function which allocates a stack frame for a function application.

The construction of a frame can be identified by the function component name of the function application *closure*, pointed to by the root below the argument pointers in a frame context. This can be achieved by tagging the heap cells representing the application nodes with a function name. Our heap profiling tool does precisely this (see Section 4.2). Thus, we obtain the construction of a stack frame from the construction tags of heap cells. The producer of a frame is identified by the top-level function which allocates a frame for a function application. And so tagging frames is performed as follows.

We extend every stack frame with a *marker* which points to profiling information identifying the producer and the construction. When a new stack frame is allocated, the marker is pushed on top of the frame and permanently stored until the frame is popped off the stack. As a result, the construction of every frame is identified by the name of the executing function *regardless of the frame context*. For example, a frame is tagged with the name of a corresponding function, even when the function application has been computed and the root of the function application is updated with a result.

A downside of this tagging scheme is that a frame may need to be re-tagged in the presence of tail-call optimisation: a tail-call re-uses the old stack frame to represent another call.

5.4 Implementation

Our implementation of Pure has an instrumented compiler and runtime system for a stack profiler. The modifications of the compiler for stack profiling are similar to the instrumentation for heap profiling, as described in Section 4.2.2.

There are also additional modifications to the runtime system to profile the stack. We extend every stack frame with a *marker* to accommodate *static* tags which carry information about stack frames determined at compile-time. Space is reserved for a marker which points to the profile information supplied by the compiler. For example, Figure 5.3 demonstrates how a stack frame for the function application `(sum 100 50)` is tagged. Static tags identify two kinds of information: the function that produced the stack frame and what construction it represents.

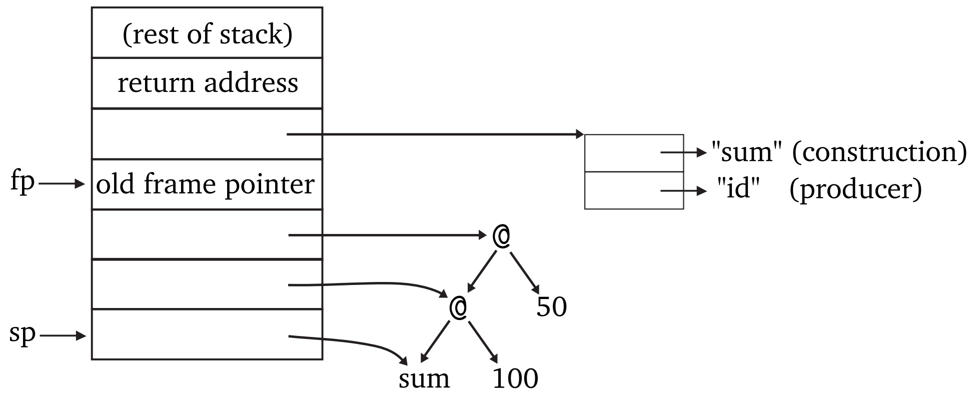


Figure 5.3: An example of a tagged stack frame.

In our implementation, EVAL is the only instruction which allocates a stack frame (see Section A.2.2 of Appendix A). So marking frames with static tags is performed by *instrumenting* EVAL as follows. When a function call is made, the EVAL instruction allocates a new stack frame, pushing a marker associated with the construction of the function call and the producer of the allocated stack frame.

Attributing the cost of a stack frame to a producer or a construction is performed naturally by traversing the stack for samples in a top-bottom fashion. Garbage collection provides a natural opportunity to traverse the stack for a sample. The cost of sampling is reduced by combining garbage collection and stack sampling. Additionally, this provides consistency between stack and heap profiles. The runtime stack is sampled every b bytes of *stack allocations*, where b is defined when a program is executed. This way, increasing the interval frequency allows previously hidden stack spikes to be captured in a profile.

Of course, before the runtime stack is traversed, the graph on top of the stack must be in a proper form for the sampling results to be consistent; the nodes forming the graph of a function application are constructed in the heap and there *must* be no stack frames for a weak-head-normal node on top of the stack. Sampling the stack is performed when code for a function is *entered*, right before the ENTER instruction is executed (see Section A.2.2 of Appendix A). This ensures there is no frame for a WHNF node on top of the stack and that all of the roots of the function application are present on top of the runtime stack. And so sampling works as follows.

We extend the runtime system of Pure with an allocation meter which ticks at stack allocations; the meter value increases by the size of each new stack element in bytes. Before a function is entered, a test is performed by a meter-handler which detects if a sampling interval has elapsed. If so, a sampler is invoked which suspends graph reduction for a *census* of the stack, garbage collection is performed and the stack is traversed to collect population counts of stack frames by aggregating the bytes associated with each stack item of a stack frame. After stack traversal is complete, a census is recorded in a log file. The implementation resets the meter and execution is resumed for another sample.

Our implementation uses the *semispace copying* garbage collection algorithm of Cheney [72, Chapter 4], extended to cope with frames markers. The stack traversal for a census is performed during the *evacuating* phase of the garbage collector; population counts of stack frames are collected when the heap cells pointed to by the stack are evacuated from the old heap (*from-space*) to the new heap (*to-space*), ignoring the space occupied by the markers in the stack. Consequently, the logged censuses exclude profiling overheads.

The censuses themselves are stored in a text file with a similar structure to files recording heap censuses. A census consists of a header identifying the stack allocation count at which the census was taken in bytes, followed by *census records*. Each census record comprises a *marker* and corresponding stack-frame population count in bytes. The marker associates frame population counts with the *static attributes* of stack frames (i.e. producer and construction) in a table record. For example, the census records

```
CENSUS 8000
  12 880
  16 692
  22 244
```

report 880 bytes occupied by the stack frame annotated with the marker 12, 692 bytes by the stack frame with a marker 16 and 244 bytes attributed to the stack frame marked with 22.

A separate program (ME2G in Appendix B) *post-processes* the census file to generate a stack profile (or *chart*) in a graphical form, particularly *producer*, *construction* profiles and their *restrictions*. Note that, because we have preserved the stack frame markers and the information that they point to (in a profile table), these profiles can be generated without re-executing a program for profiling. We also provide *hotspot* profiling for the stack.

Our implementation of stack profiling is rather reminiscent of a *heap profiler*. Indeed, we have re-used profiling techniques from Chapter 4 for sampling the runtime stack.

5.5 Results

In this section we apply stack profiling to a series of programs written in the *Pure* language. We conclude with an evaluation of stack profiling.

5.5.1 The sumChops program

We begin with the sumChops program in Figure 5.4. The sumChops program is defined in terms of the function sumChops which takes an integer n as its first argument and a list of positive integers xs as its second argument. The list xs is chopped into sections, each of n elements. The result is a list of section sums. For example, consider the following expression:

```
sumChops 2 (Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil))));
```

```
mkList f t
  | f == t = Cons f Nil
  | f <  t = Cons f (mkList (f+1) t);

take n Nil = Nil;
take n (Cons x xs)
  | n == 0    = Nil
  | otherwise = Cons x (take (n-1) xs);

drop n Nil   = Nil;
drop n (Cons x xs)
  | n == 0    = Cons x xs
  | otherwise = drop (n-1) xs;

chop n xs = case xs of {
              Cons y ys -> Cons (take n xs)
                                (chop n (drop n xs));
              Nil       -> Nil };

foldr xs = foldrSum 0 xs;

foldrSum e Nil          = e;
foldrSum e (Cons x xs) = sum x (foldrSum e xs);

sum x y = x + y;

sumChops n xs = sumChop (chop n xs);

sumChop Nil          = Nil;
sumChop (Cons x xs) = Cons (foldr x) (sumChop xs);

main = sumChops 500 (mkList 1 1000)
```

Figure 5.4: The sumChops program. The function chop is based on the function chop8 in [2, Section 7.6].

Here, the function `sumChops` chops the list into two sections, the first containing the first two elements of the list and the second containing the last two elements of the list. Each section is summed to give a list of the sums. The value of the expression above is therefore: `Cons 3 (Cons 7 Nil)`.

For this illustration, we use the function `mkList` to produce a list of one thousand elements and the function `sumChops` to chop this list into two parts.

The first stack profile that we produce is the construction profile of Figure 5.5. The stack grows, reaching a peak of about 12 kb twice, over approximately 60 kb of stack allocations. Most of the accumulation of stack space is associated with the application of the function `foldrSum`. Who produces the stack frames for the applications of the function `foldrSum`? The producer profile in Figure 5.6 shows it is the top-level function `sum`.

Investigating the definitions of the functions `foldrSum` and `sum`, the problem is apparent.

```
foldrSum e Nil          = e;
foldrSum e (Cons x xs) = sum x (foldrSum e xs);

sum x y = x + y;
```

The function `sum` is strict in its arguments. So when the function application of `sum` in the second equation of `foldrSum` is applied, the function `sum` allocates a stack frame (by the second argument of the + operator) for the evaluation of the expression `(foldrSum e xs)`. This cycle is repeated recursively, pushing a series of frames on the stack for the evaluation of the function `foldrSum` to sum a list proportional to the elements of the list argument. The summation of the list is accumulated and not computed until the base element of the function `foldSum` is applied to the last summation (from the right-hand side). Meanwhile, `foldrSum` retains its arguments. The first spike in Figure 5.5 is associated with the folding of the first section of the list and the second spike is associated with the folding of the second section of the list.

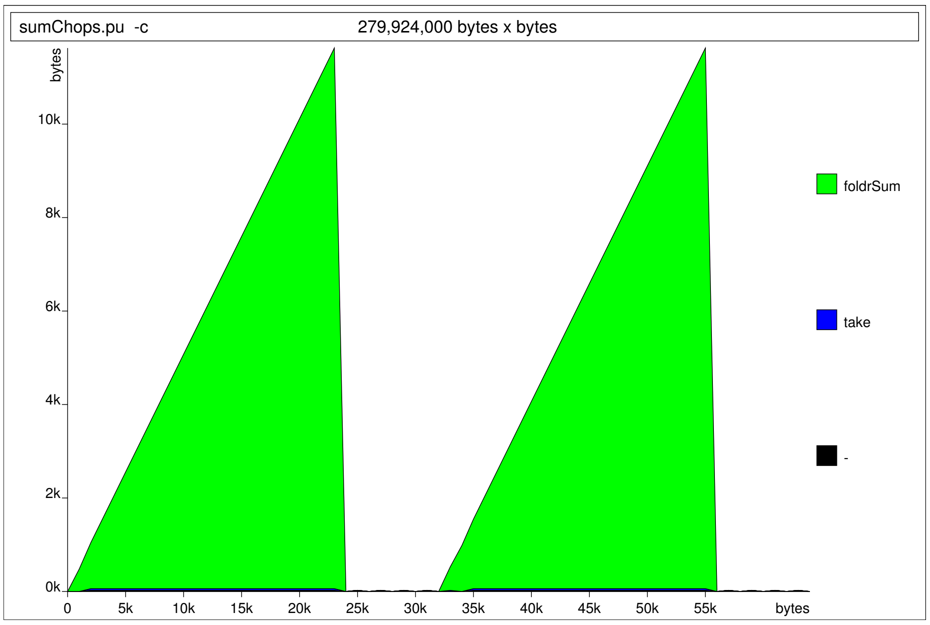


Figure 5.5: A construction profile for `sumChops`.

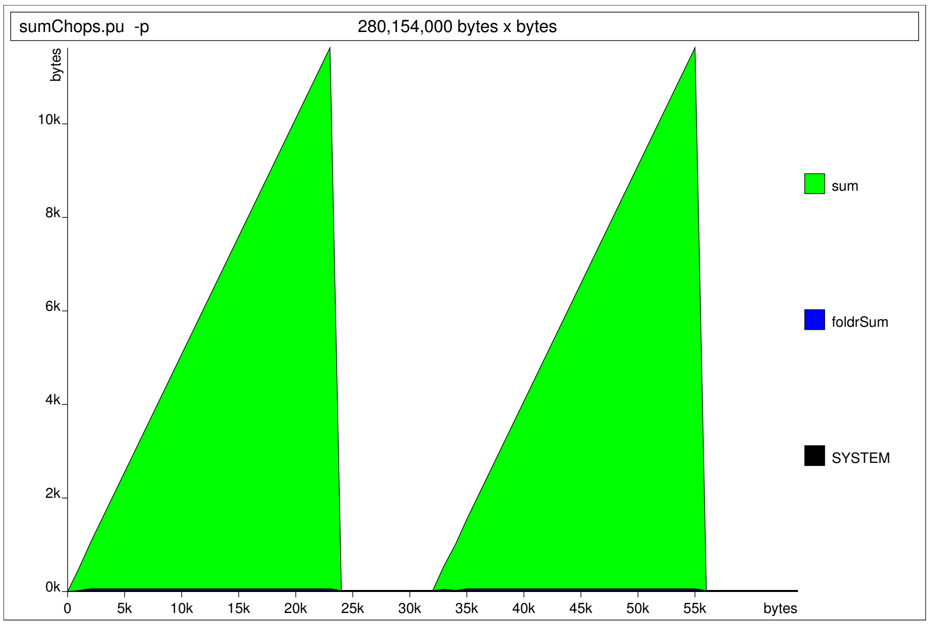


Figure 5.6: A producer profile for `sumChops`.

Folding a list from right to left using a strict function is a pitfall; it requires the elements of an entire list to be processed before returning a value. In such situations, a better technique is to fold from left to right using an accumulator for the summation of the list. The function `foldl` is the fold version which achieves this:

```
foldl xs = foldlSum 0 xs;

foldlSum e Nil          = e;
foldlSum e (Cons x xs) = foldlSum (sum e x) xs;
```

The construction and producer profiles after replacing the function `foldr` with `foldl` are shown in Figure 5.7 and Figure 5.8, respectively.

The overall area representing both stack depth and duration is reduced but the maximum stack depth is almost unchanged. Part of the previous problem remains: the function `foldlSum` accumulates nested applications of suspended summation. Not until the function `foldlSum` returns can the evaluation of the nested applications of the function `sum` begin, causing a spike in the stack (see Figure 5.7) for the summation of each section of a chopped list. One remedy is to force summations as they are accumulated, using the built-in function `seq`:

```
foldlSum e Nil          = e;
foldlSum e (Cons x xs)
    = let summed = sum e x
      in seq summed (foldlSum summed xs);
```

The function `seq` forces the evaluation of its first argument and returns its second argument. As a result, the sum of a previous value and a list element are evaluated at each iteration of the function `foldlSum`. A stack profile after the above modification applied to the scale of the previous profile is shown in Figure 5.9. The stack peak is reduced from 12 kb to about 0.1 kb. The stack (and the heap) consume constant memory.

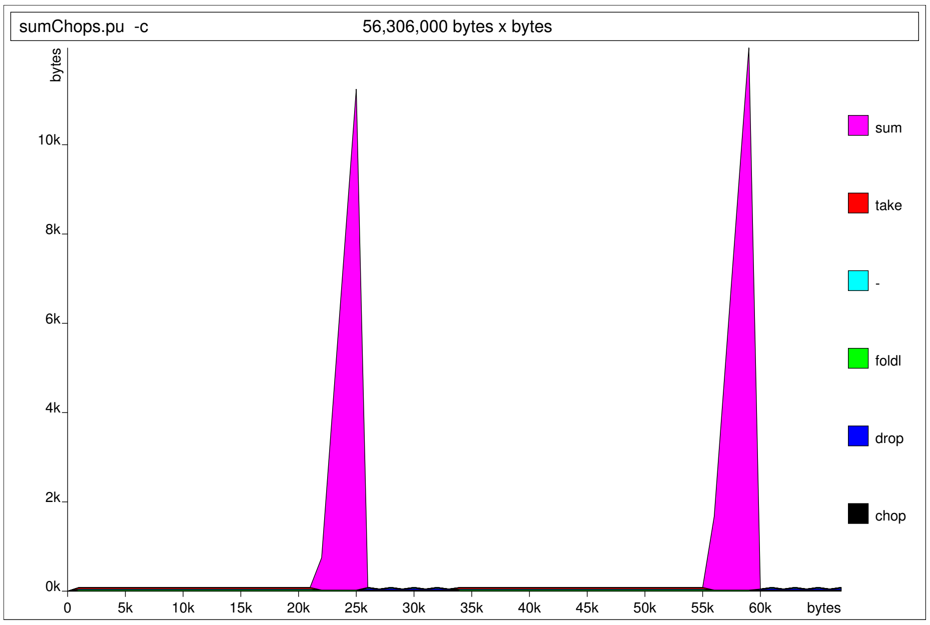


Figure 5.7: A construction profile after introducing the function `foldl`.

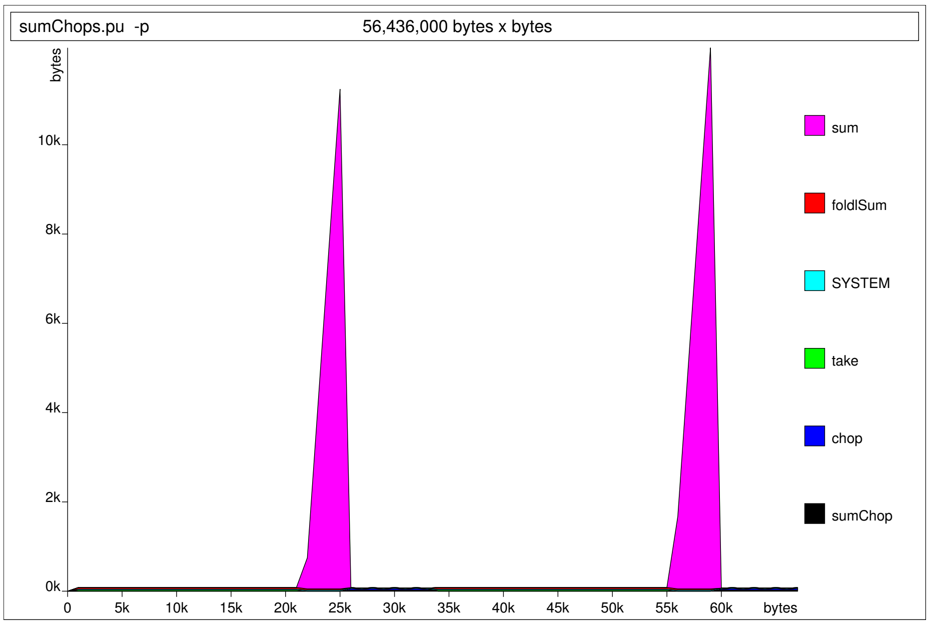


Figure 5.8: A producer profile after introducing the function `foldl`.

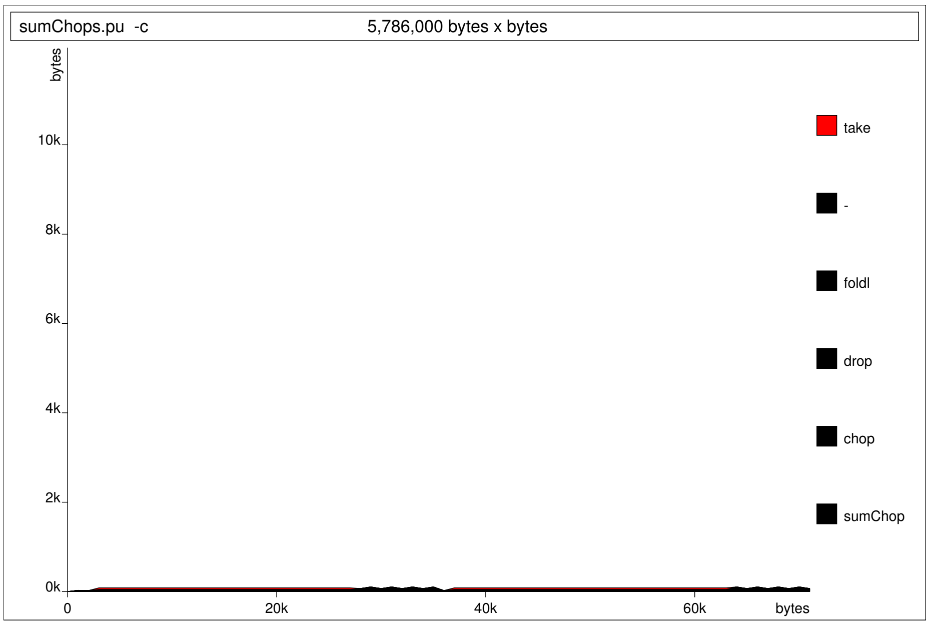


Figure 5.9: A construction profile after strictifying the application of `sum`.

5.5.2 The `queens` program

The next example is the `queens` program in Figure 5.10. What follows is a brief summary describing the `queens` program. It is closely based on the summary in [5, Chapter 6].

The `queens` program implements the *n-queens* problem. Given a chessboard and `n` queens, they are placed on the chessboard so that none of the queens hold each other in check; that is, two queens may not lie in the same column, row or diagonal. The `queens` program solves this problem as follows.

A queen is placed in the first column. Then a queen is placed in the second column at a position not held in check by the first queen. Then a queen is placed in the third column at a position not held in check by the second and first queens. This process is continued until all `n` queens have been placed on the chessboard. This algorithm is repeated with backtracking to find all of the possible solutions.

A chessboard is represented by a list of `n` columns, giving for each column the row in which a queen appears. For example, the list:

```
Cons 4 (Cons 8 (Cons 3 (Cons 1 Nil)))
```

represents a chessboard with a queen in the fourth row of the first column, eighth row of the second column, third row of the third column, and first row of the fourth column.

At the heart of the program is the function `nsoln`, which takes a number of queens as its argument (nine in this example) and returns the number of possible solutions (yielding 352). The function `gen` generates a list of queens solutions through the functions `concatMapg1` and `concatMapg2`. Finally, the function `safe` implements the check that a new queen is not in conflict with any queen already on the board.

For the purpose of this illustration, heap and stack profiles are both used. The aim is to reduce the space needed for both stack and heap, with a focus on stack space.

```
and False a = False;
and True  a = a;

append Nil         ys = ys;
append (Cons x xs) ys = Cons x (append xs ys);

concatMapg2 b Nil         = Nil;
concatMapg2 b (Cons x xs) = append (genq b x)
                                   (concatMapg2 b xs);

concatMapg1 nq Nil         = Nil;
concatMapg1 nq (Cons x xs) = append (concatMapg2 x (toOne nq))
                                    (concatMapg1 nq xs);

length Nil         = 0;
length (Cons x xs) = 1 + length xs;

nsoln nq = length (gen nq nq);

gen nq n = case (n == 0) of {
           True -> Cons Nil Nil;
           False -> concatMapg1 nq (gen nq (n - 1))
           };

genq b q = case (safe q b) of {
             True -> Cons (Cons q b)  Nil;
             False -> Nil
           };

safe q b = safe' q 1 b;

safe' x d Nil = True;
safe' x d (Cons q l) =
  and (x /= q) (
  and (x /= (q + d)) (
  and (x /= (q - d)) (
  safe' x (d + 1) l)));

toOne n = case (n == 1) of {
          True  -> Cons 1 Nil;
          False -> Cons n (toOne (n - 1))
          };

main = nsoln 9
```

Figure 5.10: The queens program.

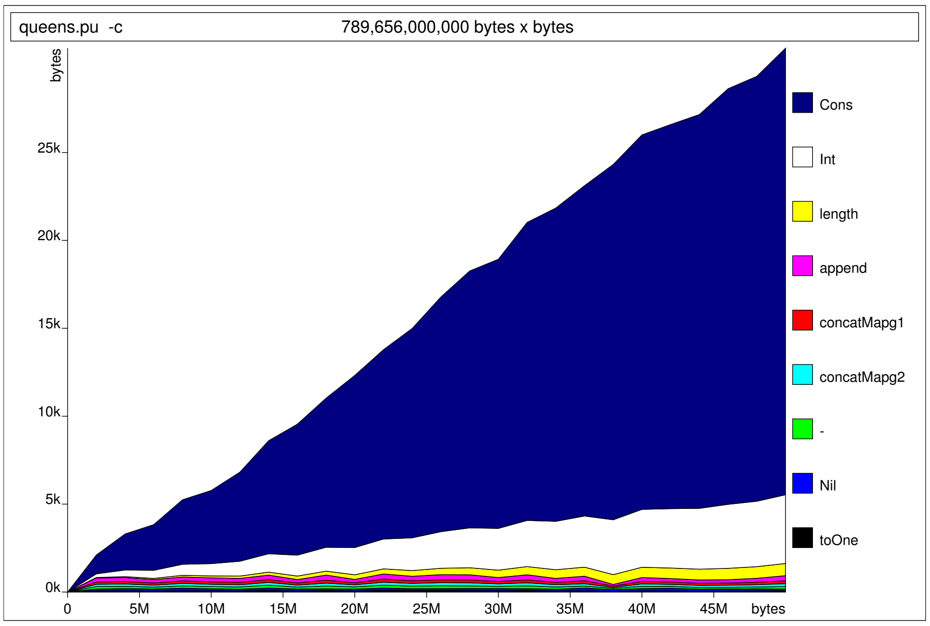


Figure 5.11: A heap construction profile of queens.

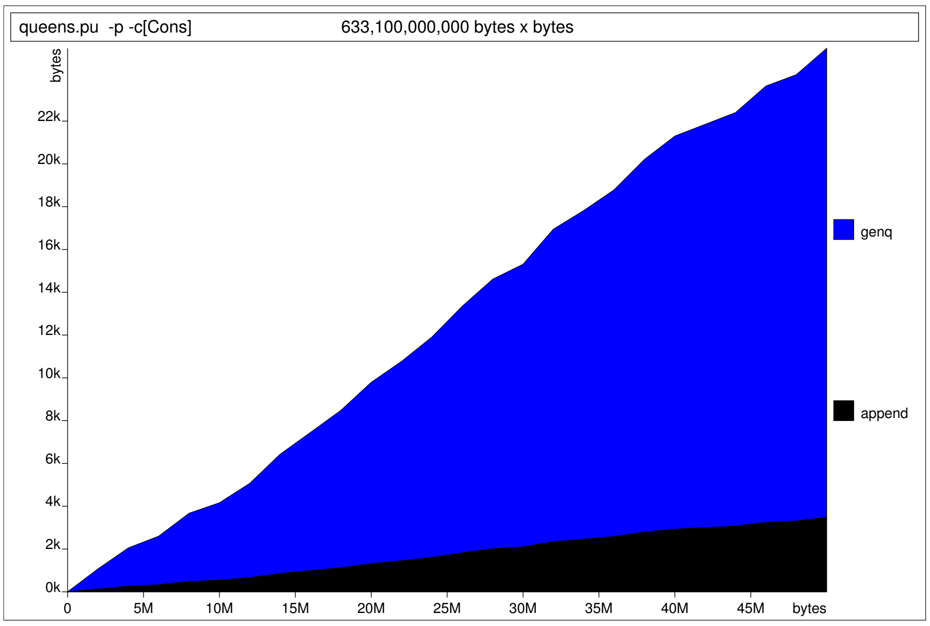


Figure 5.12: A heap producer profile restricted to Cons.

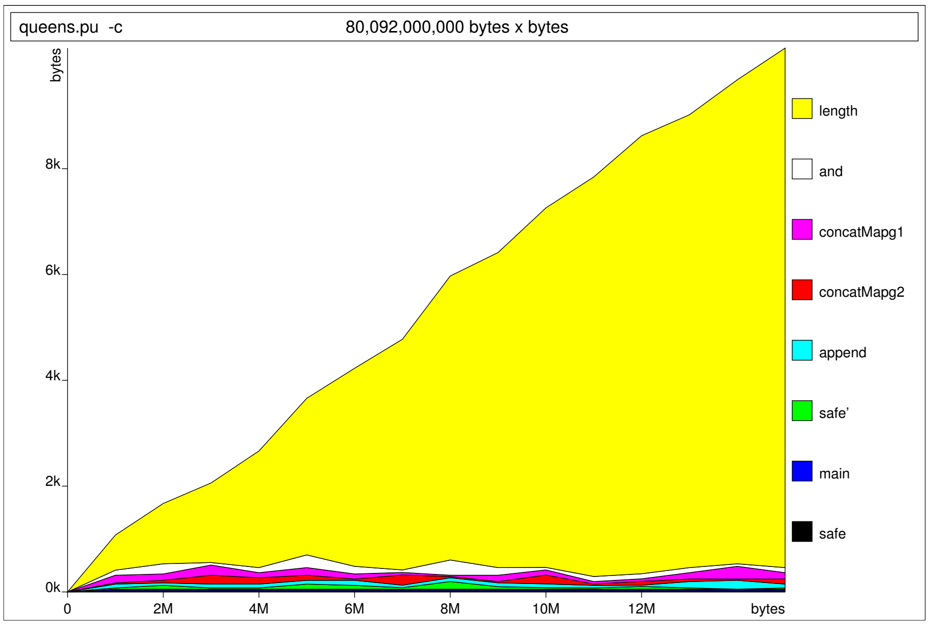


Figure 5.13: A stack construction profile of `queens`.

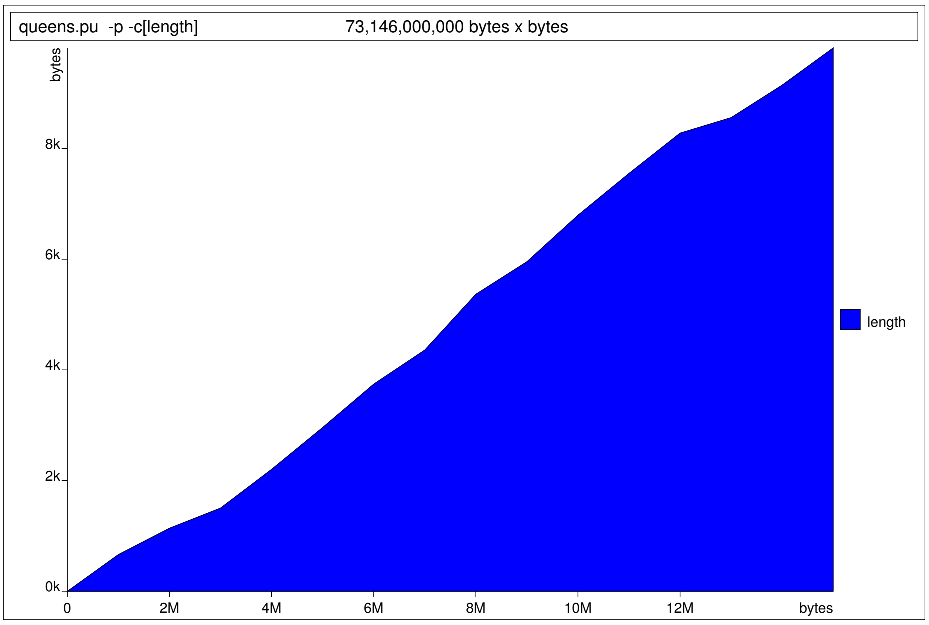


Figure 5.14: A stack producer profile restricted to `length`.

The construction heap profile of queens is in Figure 5.11. Heap space steadily increases, reaching a peak of about 30 kb in a computation involving about 50 Mb of heap allocation in total. The majority of heap space is associated with Cons. As shown in Figure 5.12, the bulk of these Conses are produced by the function genq. This information suggests that the list of queens is accumulated and retained in the heap.

Seeking further information, we apply the stack profiler. Figure 5.13 is a construction profile. Stack depth increases with a peak of approximately 10 kb in about 16 Mb of stack allocations. The majority of stack frames are for applications of the function length. The producer of these stack frames is the function length itself (see Figure 5.14). Investigating the definition of length below, a problem is apparent:

```
length Nil          = 0;
length (Cons x xs) = 1 + length xs;
```

the function length traverses the entire list to compute the number of solutions for n-queens, and at every recursive call to length, a stack frame is allocated to for the application of length, which retains its argument. From this information it seems that the function length is accumulating the Conses (Figure 5.11) in the heap. Avoiding stack allocation for length can be achieved by modifying length as follows:

```
length xs = length' 0 xs;

length' acc Nil          = acc;
length' acc (Cons x xs) = let summed = acc + 1
                          in seq summed (length' summed xs);
```

Here, length' forces the summation using seq and passes the summation as an argument. The results are shown in the stack profile of Figure 5.15 and the heap profile of Figure 5.16. Stack space is reduced from a peak of about 10 kb to approximately 0.9 kb, and heap space is reduced from about 30 kb to 1.4 kb.

Looking at the heap producer profile (Figure 5.16), it seems that the program's memory-consumption problems are solved. However, looking at the stack producer profile (Figure 5.15) now draws attention to the functions append and and as the main producers of stack frames.

Even though stack depth is less than 1 kb (Figure 5.15), let us reduce stack space further.

The function and appears in the definition of the function safe' (an auxiliary of safe). Examining the definition of safe', a problem is apparent:

```
safe' x d Nil = True;
safe' x d (Cons q l) =
  and (x /= q) (
  and (x /= (q + d)) (
  and (x /= (q - d)) (
  safe' x (d + 1) l)));
```

the function safe' recursively calls itself if x and q do not conflict. At each recursive call the function and allocates a stack frame, increasing stack depth. One way to reduce stack depth is to write a tail-recursive version of safe', replacing every rule with an auxiliary function as follows.

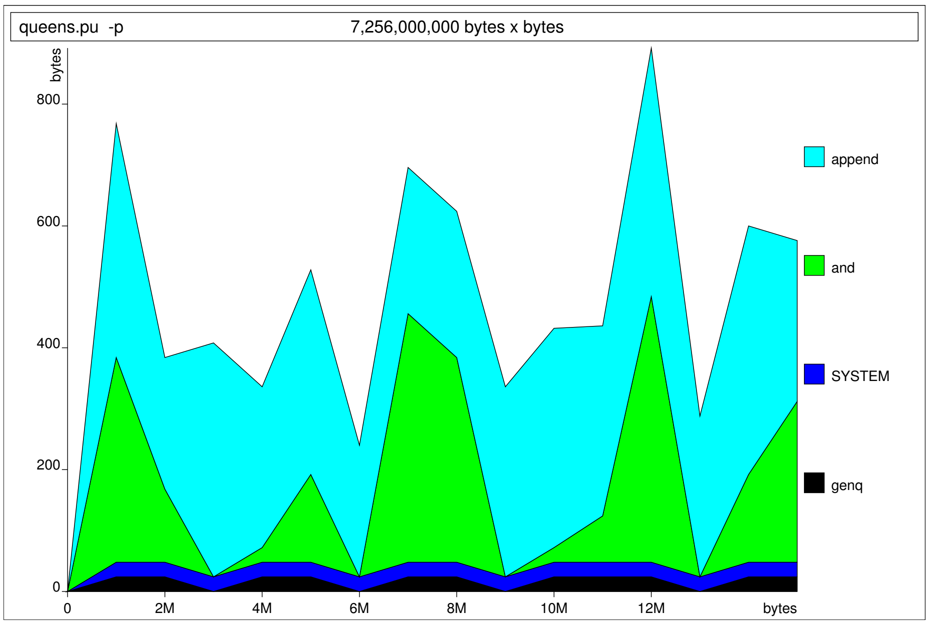


Figure 5.15: A stack producer profile after introducing `length'`.

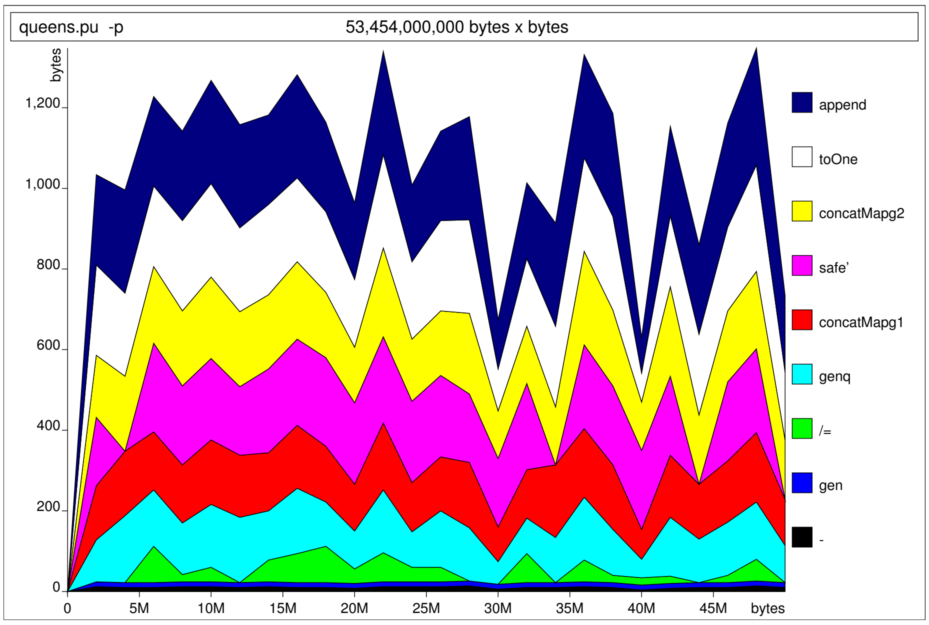


Figure 5.16: A heap producer profile after introducing `length'`.

```
safe' x d Nil = True;
safe' x d (Cons q l) = diagonalL x d q l;

diagonalL x d q l = if x /= (q - d) then diagonalR x d q l
                    else False;

diagonalR x d q l = if x /= (q + d) then sameRow x d q l
                    else False;

sameRow x d q l = if x /= q then safe' x (d + 1) l
                  else False;
```

The new stack producer profile on the same scale as the previous one is in Figure 5.17. Stack space is reduced from a peak of about 0.9 kb to a peak of approximately 0.6 kb.

The problem is now clearly the remaining stack space produced by `append` which appears in the definitions of `concatMapg1` and `concatMapg2`. The function `append` is used to implement the "list of successes" technique for the possible solutions of n-queens [75]; for example, a failure of a queen insertion is represented by `Nil`. However, this can be inefficient, as it requires concatenating every solution: the construction `Nil` is used only to be discarded by `append`. It is more efficient to construct the list of solutions as they are generated. One way of achieving this is to merge `concatMapg2` and `genq`, and to replace the calls to `append` by continuations as follows.

```
concatMapg2 b Nil         nq ys = concatMapg1 nq ys;
concatMapg2 b (Cons x xs) nq ys =
 case (safe x b) of {
 True  -> Cons (Cons x b) (concatMapg2 b xs nq ys);
 False -> concatMapg2 b xs nq ys
 };

concatMapg1 nq Nil = Nil;
concatMapg1 nq (Cons x xs) = concatMapg2 x (toOne nq) nq xs;
```

The new stack profile is the producer profile in Figure 5.18 shown on the same scale as the previous stack profile. The final result is a reduction of stack space from approximately 10 kb to about 0.2 kb. The heap profile is shown in Figure 5.19. Reducing stack space also led to a reduction in heap space from approximately 30 kb to about 0.6 kb.

From these memory profiles it seems that there could even be scope for yet further memory reduction, for example, by *sharing* the list that is produced by the function `toOne` shown in the heap profile of Figure 5.19.

The final version of `queens` takes much less memory and it executes much *faster*.

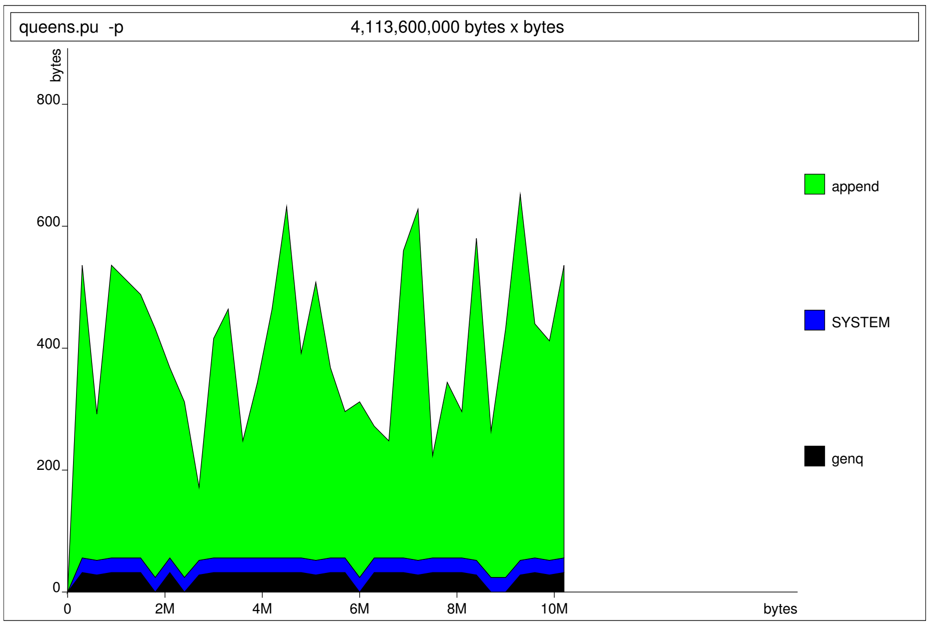


Figure 5.17: A stack producer profile after modifying `safe'`.

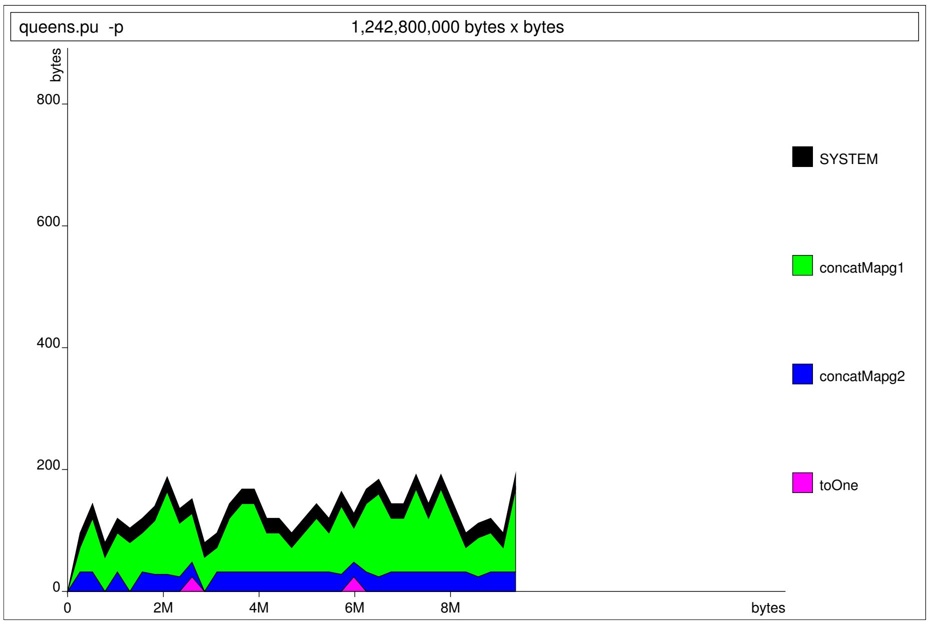


Figure 5.18: A stack producer profile after removing `append`.

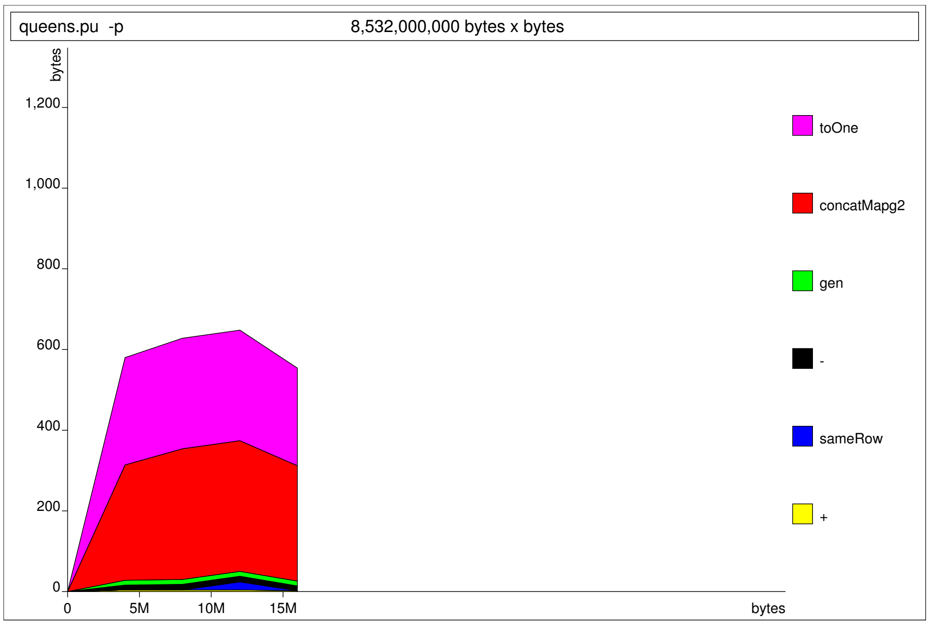


Figure 5.19: A heap producer profile after removing `append`.

5.5.3 Evaluation

We have applied the stack profiler to observe the stack memory behaviour of two programs: the `sumChops` program in Section 5.5.1 and the `queens` program in Section 5.5.2. For the `sumChops` program, stack space is reduced from a peak of approximately 12 kb to about 0.1 kb. For the `queens` program, stack space is reduced from a peak of about 10 kb to approximately 0.2 kb.

The method of stack profiling is similar to heap profiling (e.g. in [1]): profiling is iteratively applied to obtain stack frame producers, construction or their combinations.

Stack profiles are simpler than heap profiles, since stack contents are closures and heap contents are closures and constructors. Thus, a stack profile can helpfully draw a programmer's attention to components with high demands for memory in a simpler way than a heap profile. However, the simplicity of stack profiles can also mean that they lack information that may sometimes be necessary to understand the cause of excessive stack demands.

Stack profiles can shed light on space leaks that are not fully apparent in heap profiles. An example of this is the indirect influence on heap memory by the function `length`, whose application frames occupied the stack (see Figures 5.11 and 5.14). Similarly, heap profiles can shed light on space leaks that are not fully apparent in stack profiles. An example of this is the space leak caused by the function `toOne` (see Figures 5.18 and 5.19).

A good technique is to obtain both heap and stack profiles. Each profile may point to different sources of space leaks.

In Section 5.5.2, a dramatic reduction in stack memory also led to a dramatic reduction in heap memory. However, this is not always the case. After reducing the stack memory of the `compile` program (which is a version of the arithmetic code evaluator in [2, Chapter 13]) to about a peak of 80 b (see Figure 5.20), a space leak remained in heap memory (see Figure 5.21).

The GHC heap profiling system, a state-of-the-art compiler for Haskell, provides an option to include stack space in heap profiles. However, stack memory is aggregated to a single undifferentiated category in memory which bears no relation to the source program. An example is the construction profile in Figure 5.22, stack space is associated with the key `STACK`.

Our construction profiles for the stack cannot distinguish between different instances of function applications. For example, a construction profile of the `queens` program reported the function `append`. In the source of the `queens` program, `append` is only used in the body of two functions (other than `append` itself). However, if `append` is used in many other functions we have no ready way of distinguishing which instance of `append` is causing excessive stack demands, other than calling the `append` function under different names.

So far, stack profiling has been illustrated for small programs. However, in Chapter 6 a larger program is used to evaluate stack profiling further.

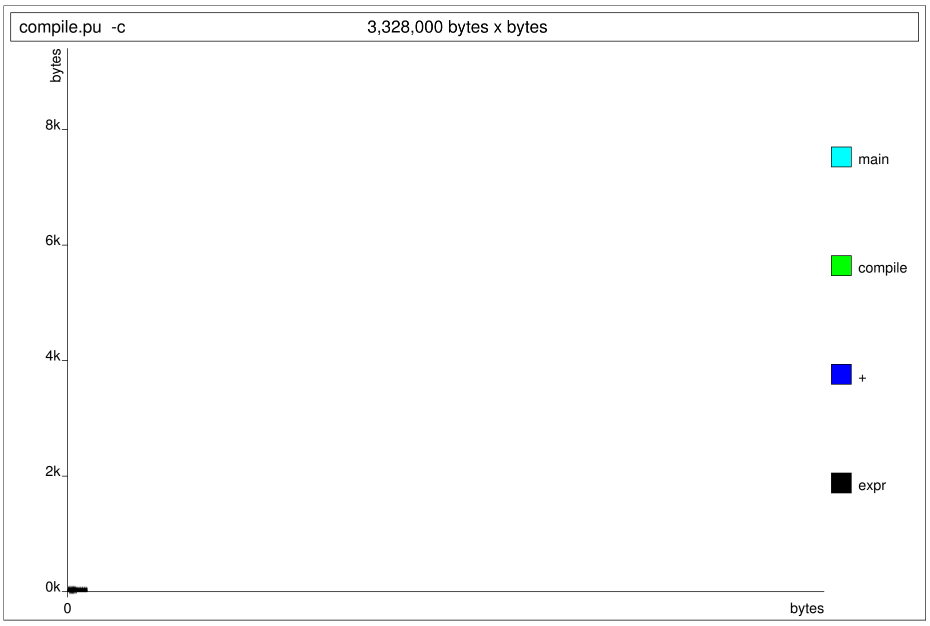


Figure 5.20: A stack construction profile of `compile`.

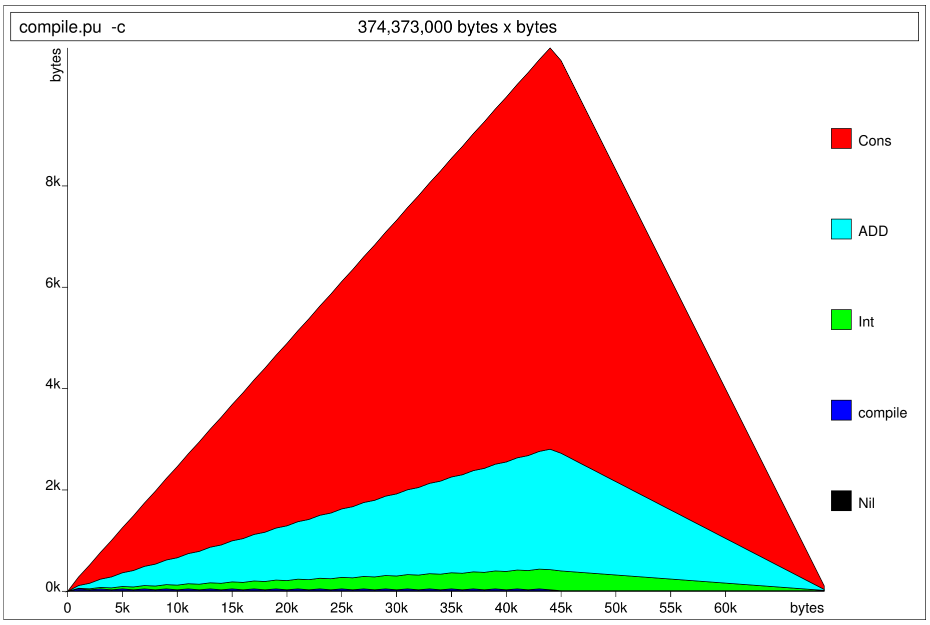


Figure 5.21: A heap construction profile of `compile`.

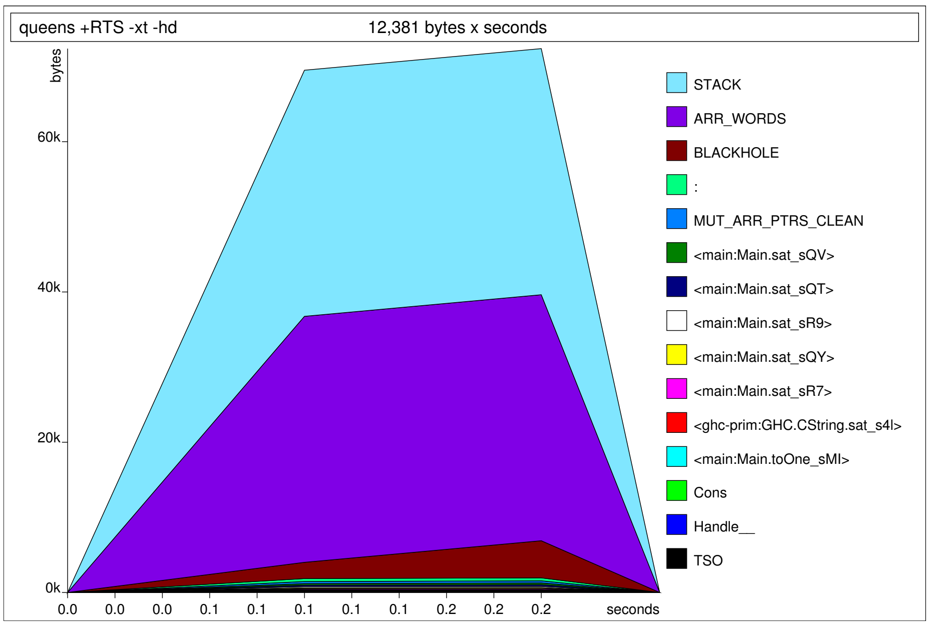


Figure 5.22: A construction profile produced using the GHC (version 7.4) profiling system with stack space included. This profile is of the `queens` program with 10 queens as input.

Chapter 6

Clausify Revisited

In this chapter we briefly revisit the `clausify` program to further demonstrate the use of our *hotspot* (Chapter 4) and *stack* (Chapter 5) profiling systems. Section 6.1 sets out our motivation for profiling `clausify`. Section 6.2 describes the `clausify` program in summary. Section 6.3 shows how stack and hotspot profiles highlight the major space leaks of `clausify`. Section 6.4 evaluates our profiling results.

6.1 Motivation

The `clausify` program was originally used by Runciman and Wakeling to introduce heap profiling by *producer* and *construction* profiles [1], Runciman and Röjemo to present *lifetime* and *retainer* profiles [17], Sansom to demonstrate the results of *cost-centre* profiling [7], and Jarvis to evaluate the results of *cost-centre-stack* profiling [8]. Furthermore, `clausify` appears in the standard `noFib` benchmark suite [76]. And so the `clausify` program serves as an interesting example of stack, and in particular, hotspot profiling.

Such extensive profiling of `clausify` led to a great reduction of memory demands (and execution time). Runciman and Wakeling used producer and construction profiles, leading to a heap memory reduction from 1.3 Mb to 7 kb [1]. Subsequently, Runciman and Röjemo used lifetime and retainer profiles, leading to a further heap memory reduction from 7 kb to 1 kb [17]. However, we shall see in Section 6.3 how the information supplied by the hotspot profiler brings an insight that might have led to such a space reduction sooner.

6.2 The `clausify` Program

In order to make this dissertation self-contained, we give a brief summary of the `clausify` program. It is closely based on the summary in [1].

The `clausify` program takes a series of propositional formulae and yields their clausal form equivalents. The transformation of a proposition to a set of clauses can be specified by the following rules:

- `elim` eliminates equivalence and implications:

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

- `negin` makes negations the innermost connectives:

$$\begin{array}{rcl} \neg\neg p & \rightarrow & p \\ \neg(p \vee q) & \rightarrow & \neg p \wedge \neg q \\ \neg(p \wedge q) & \rightarrow & \neg p \vee \neg q \end{array}$$

- `disin` pushes disjunctions within conjunctions:

$$\begin{array}{rcl} p \vee (q \wedge r) & \rightarrow & (p \vee q) \wedge (p \vee r) \\ (p \wedge q) \vee r & \rightarrow & (p \vee r) \wedge (q \vee r) \end{array}$$

- `split` splits up conjuncts:

$$\begin{array}{rcl} p \wedge q & \rightarrow & p \\ & & q \end{array}$$

- `unicl` forms a set of unique non-tautologous clauses:

$$q_1 \vee \ \ldots \ \vee q_{\mathrm{n}} \vee \neg p_1 \vee \ \ldots \ \vee \neg p_{\mathrm{m}} \quad \rightarrow \quad (\{q_1, \ldots, q_{\mathrm{n}}\}, \{p_1, \ldots, p_{\mathrm{m}}\})$$

  A clause (qs, ps) is tautologous if $(qs \cap ps) \neq \emptyset$.

The abstract syntax of propositional formulae is represented by the `Formula` data type. Note that we have used `Int` for symbols to represent characters:

```
data Formula = Sym Int
             | Not Formula
             | Dis Formula Formula
             | Con Formula Formula
             | Imp Formula Formula
             | Eqv Formula Formula;
```

The heart of the program is a "pipeline" which combines the transformation rules using several functions. Each function corresponds to one of the rules listed above:

```
clauses p =  unicl
            (split
            (disin
            (negin
            (elim p))));
```

For the purpose of profiling (see Section 6.3), we use the following proposition from Runciman and Wakeling [1] so that our profiles are comparable:

$$(a = a = a) = (a = a = a) = (a = a = a)$$

Although the transformation rules reduce this proposition to the single clause $(\{a\}, \emptyset)$, the intermediate formulae involved in the transformations are extensive.

The source listing for `clausify` is contained in Appendix C. This version is based on version 0 listed in [1]. Our version of `clausify` is modified to first-order applications and simplified to exclude parsing propositional formulae and formatting output results (clausal axioms). This simplification enables `clausify` to be accepted by the Pure implementation.

6.3 Comparing `clausify` Memory Profiles

We now show how the stack profiler (Chapter 5) and the hotspot profiler (Chapter 4) are useful in comparison with the original *producer* and *construction* profiles [1]. The `clausify` program has been extensively profiled using heap profiling systems [1][7][17].

The first heap profiles in this illustration differ a little from those in [1] as a result of different compiler implementation techniques.

The first heap profiles for `clausify` are the producer profile in Figure 6.1 and the construction profile in Figure 6.2. Both of these profiles show *live* heap memory steadily increasing with a peak of approximately 700 kb in a computation involving about 10 Mb of heap *allocation* in total. There is an apparent accumulation of heap cells throughout the computation which are produced by functions appearing in different sites of the program. Most notable is `disin`, which takes part in the `clausify` pipeline. The first version of `clausify` suffers from space leaks and, therefore, this accumulation of heap cells is not a surprise. However, just by looking at these heap profiles alone, the problem is not fully apparent. And so we resort to a different part of memory, the stack.

Our first stack profile for the `clausify` program is the construction profile in Figure 6.3. What appeared insignificant in heap memory now appears significant in stack memory: the majority of stack frames are occupied by the function `foldrUnicl`. Stack depth increases, reaching a peak of about 160 kb for the nested applications of `foldrUnicl`. Who produces the stack frames for `foldrUnicl`?

The producer profile in Figure 6.4 shows the function `unicl'` (an auxiliary of `unicl`). The functions `unicl`, `unicl'` and `foldrUnicl` are defined as follows.

```
unicl a = foldrUnicl Nil a;

unicl' p x = let cp = clause p
             in
             if tautclause cp then x else insertPair cp x;

foldrUnicl z Nil          = z;
foldrUnicl z (Cons x xs) = unicl' x (foldrUnicl z xs);
```

Here, `foldrUnicl` demands the list of conjuncts and recursively applies `unicl'` to every element of this list from right to left, folding conjunctive propositions to their clausal forms. That is, the formulation of `foldrUnicl` and `unicl'` is *tail-strict*, causing `unicl` to demand its input before any output is given. On every iteration, a stack frame is allocated for the applications of `foldrUnicl`, leaving a pointer to the first argument of `unicl'` pending on the stack.

With this information, and looking at the `clausify` pipeline:

```
clauses p = unicl (split (disin (negin (elim p))));
```

it seems that the function `unicl`, the last transformation rule applied, accumulates the structure shown in the heap construction profile of Figure 6.2 until the computations are nearly over.

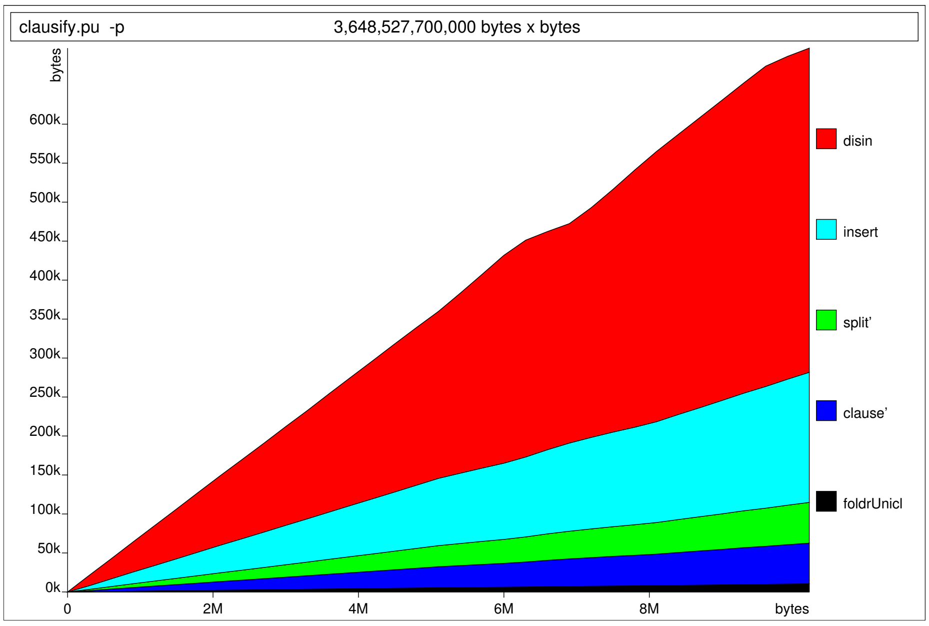


Figure 6.1: A heap producer profile of `clausify`.

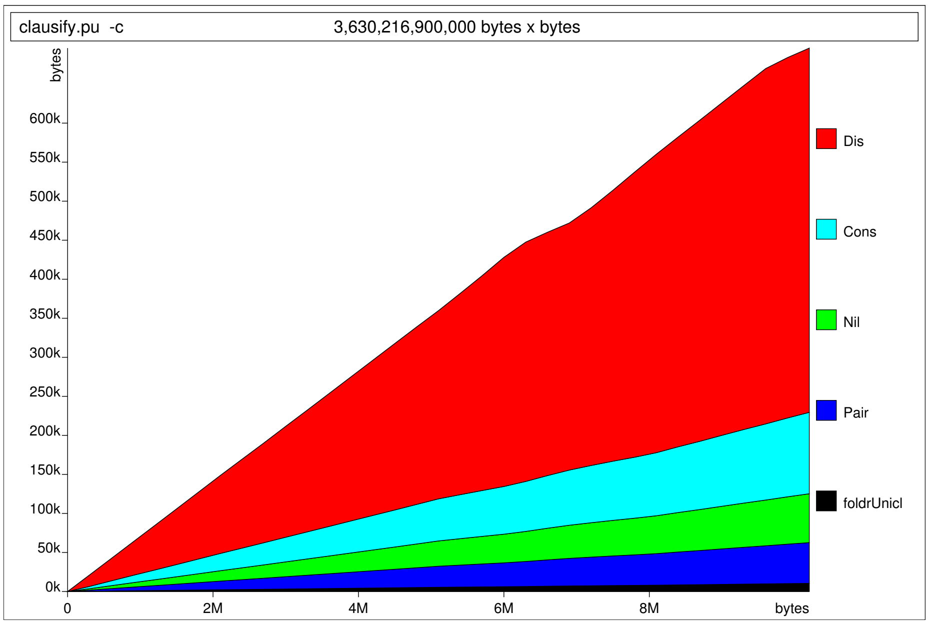


Figure 6.2: A heap construction profile of `clausify`.

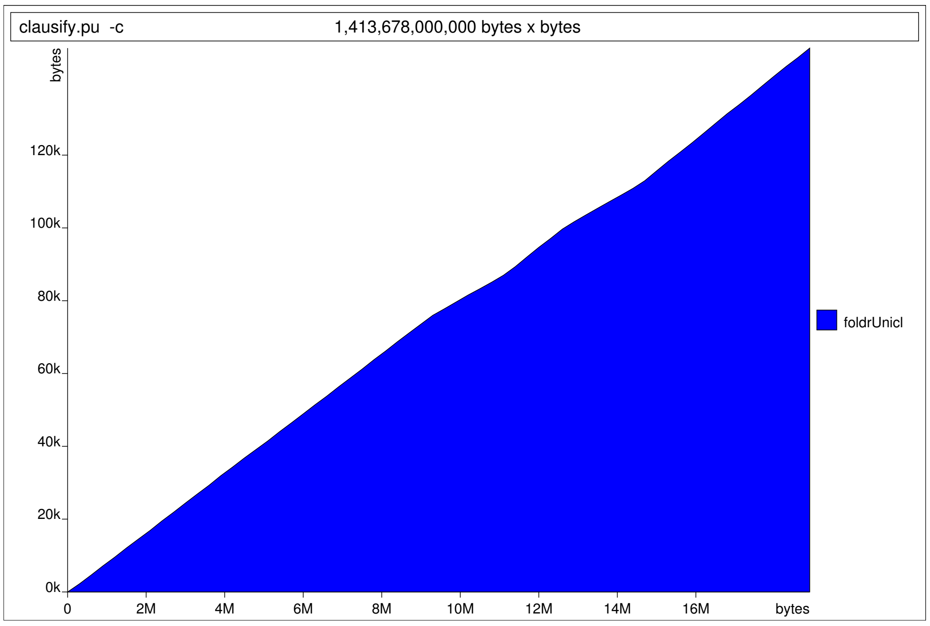


Figure 6.3: A stack construction profile of `clausify`.

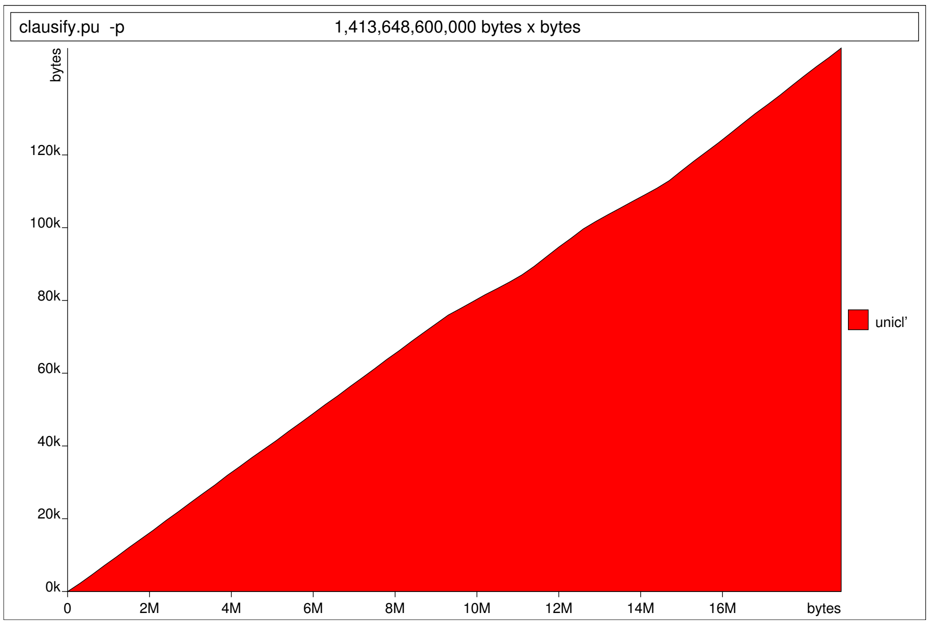


Figure 6.4: A stack producer profile of `clausify`.

Runciman and Wakeling [1] have already made the observation of the space leak caused by unicl, using their knowledge of the clausify pipeline and the output it produces. They suggest a reformulation that is not tail-strict:

```
unicl ps = filterset (mapClause ps);

filterset s = filterset' Nil s;

filterset' res Nil         = Nil;
filterset' res (Cons x xs) =
  if and (not (isElem x res)) (not (tautclause x))
  then Cons x (filterset' (Cons x res) xs)
  else filterset' res xs;
```

Note that we have modified the improved version of unicl to a first-order equivalent.

The result of the above reformulation is shown in the heap construction profile of Figure 6.5. Heap space is dramatically reduced from a peak of about 700 kb to approximately 30 kb. There is a large number of Dis constructions. Who produces these Dis constructions?

A producer profile restricted to the construction Dis shows the function disin (see Figure 6.6). Recall from Section 6.2 that the function disin distributes disjunctions over conjunctions after implications and equivalences have already been removed from the Formula. Here is how disin is defined:

```
disin prop =
case prop of {
Dis p q -> case q of {
       Con qq r  -> Con (disin (Dis p qq)) (disin (Dis p r));
       otherwise -> case p of {
                    Con pp rr ->
                    Con (disin (Dis pp q)) (disin (Dis rr q));
                    otherwise ->
                    let dp = disin p;
                        dq = disin q
                    in if or (conjunct dp) (conjunct dq)
                       then disin (Dis dp dq)
                       else Dis dp dq
                    }
       };
Con pp qq -> Con (disin pp) (disin qq);
otherwise -> prop
};
```

There is a total of six *occurrences* of Dis constructions in the body of disin. Which occurrences are associated with high memory demands? In their heap profiling experiment looking for a remedy for the space demand of Dis constructions, Runciman and Wakeling [1] make the following remark:

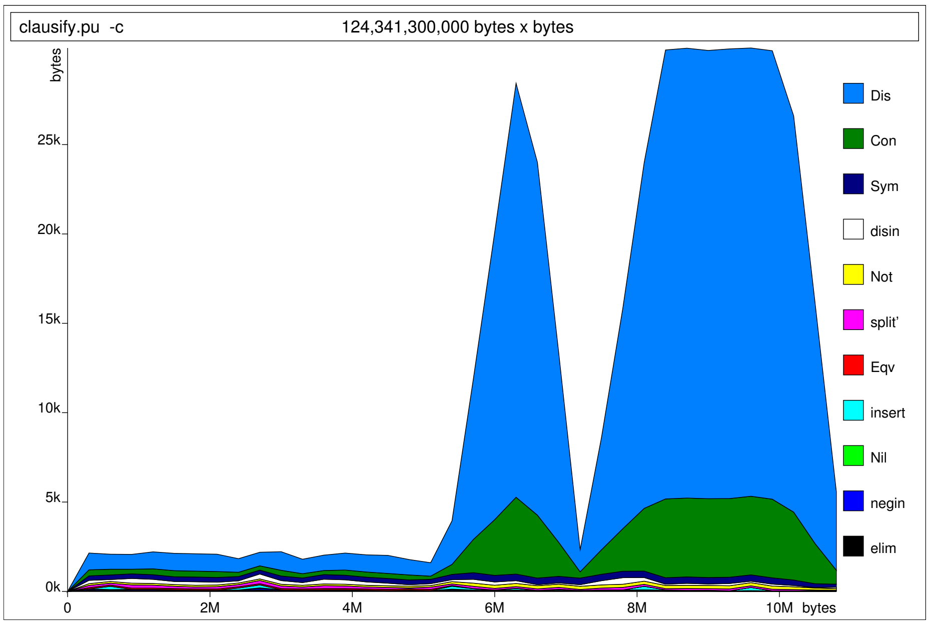


Figure 6.5: A heap construction profile after introducing `filterset`. This profile corresponds to Figure 11 of [1].

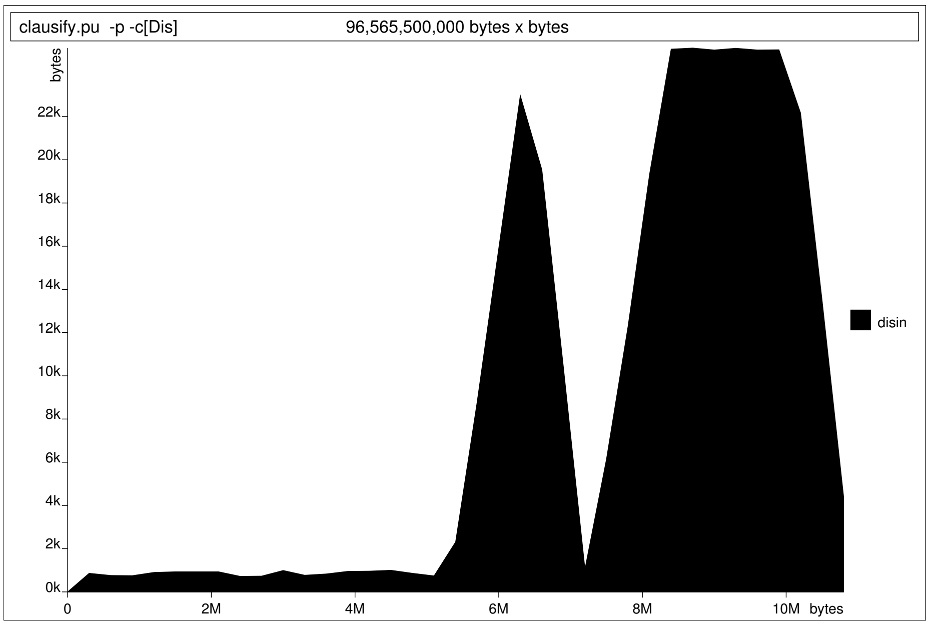


Figure 6.6: A heap producer profile restricted to the producers of `Dis`. This profile corresponds to Figure 12 of [1].

> *"Looking at the right hand sides in the full sequence of defining clauses, there are no less than six occurrences of `Dis` constructions in all. An opportunity for improvement presents itself in a notable feature common to all but the last of these: they appear as arguments to recursive `disin` calls."*

We apply the hotspot profiler. The result is shown in Figure 6.7: the hotspot profile displaying the body of `disin` specifically highlights the last `Dis` construction! The profile suggests that there is a build-up of nested `Dis` constructions which are consuming more than 40% of the heap. This leads to the intuition that `disin` traverses the `Formula` tree from the branches `dp` and `dq` of the hotspot `Dis`, applying `disin` to all of these branches before the outermost `Dis` is returned. The same problem is evident from the yellow `Con` hotspot. This information strongly suggests that `disin` is *path-strict*.

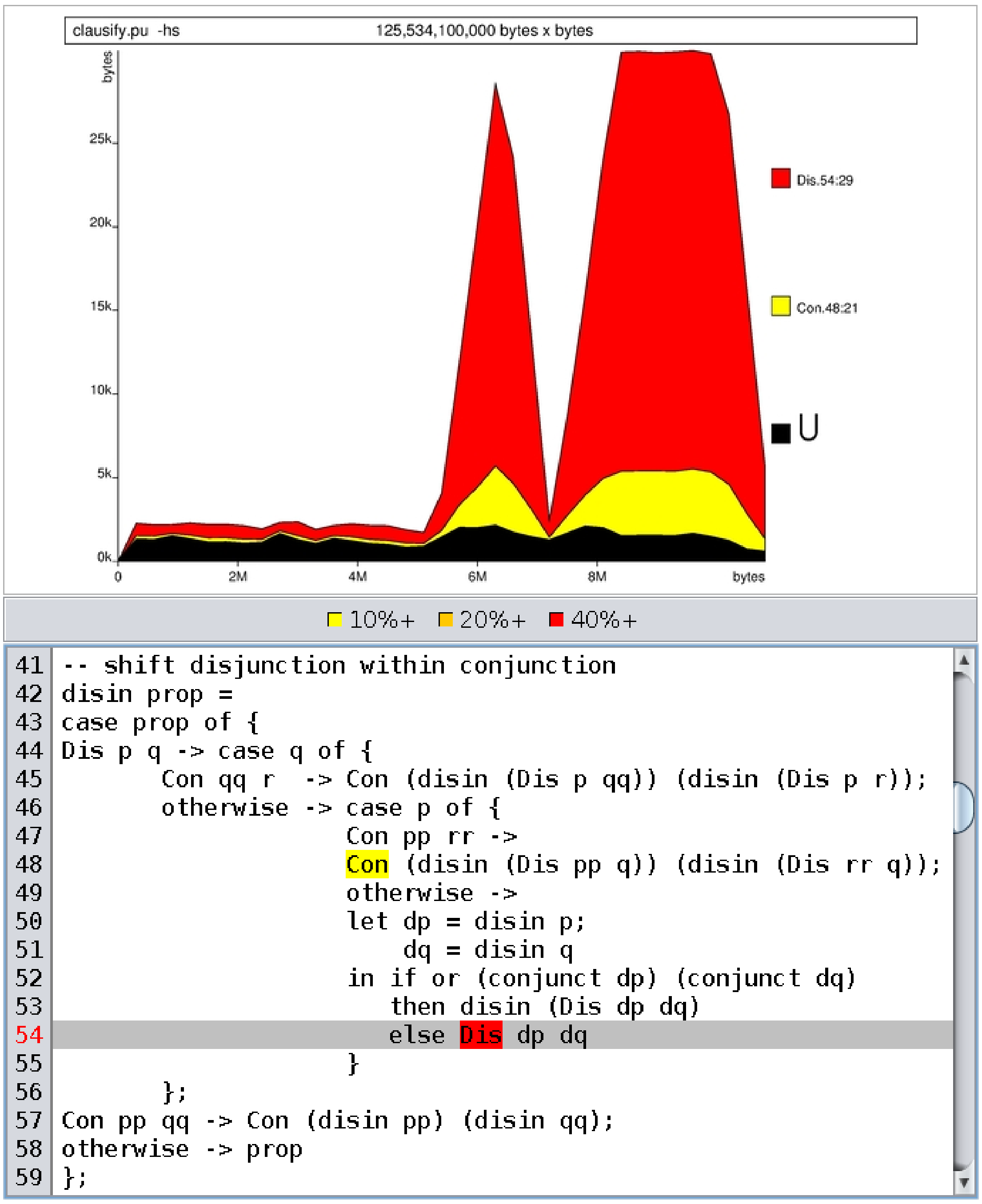
The space improving reformulation suggested by Runciman and Wakeling [1] for the function `disin` is listed below, where the explicit `Dis` constructions appearing as arguments to `disin` have been removed using the auxiliary `disin'`.

```
disin a = case a of {
          Con p q    -> Con (disin p) (disin q);
          Dis p q    -> disin' (disin p) (disin q);
          otherwise -> a
          };

disin' p q =
case p of {
Con px r  -> Con (disin' px q) (disin' r q);
otherwise -> case q of {
             Con pp qy -> Con (disin' p pp) (disin' p qy);
             otherwise -> Dis p q
             }
          };
```

The new hotspot profile is shown in Figure 6.8. The space occupied by `Dis` is reduced by almost 90% from a peak of 26 kb to a peak of 2.6 kb. However, the problem of the previous `disin` remains: the information provided by the hotspot profile in Figure 6.8 points to the same conclusion as that of the hotspot profile in Figure 6.7: `disin` is path-strict, causing an accumulation of `Dis` and `Con` constructions in the heap.

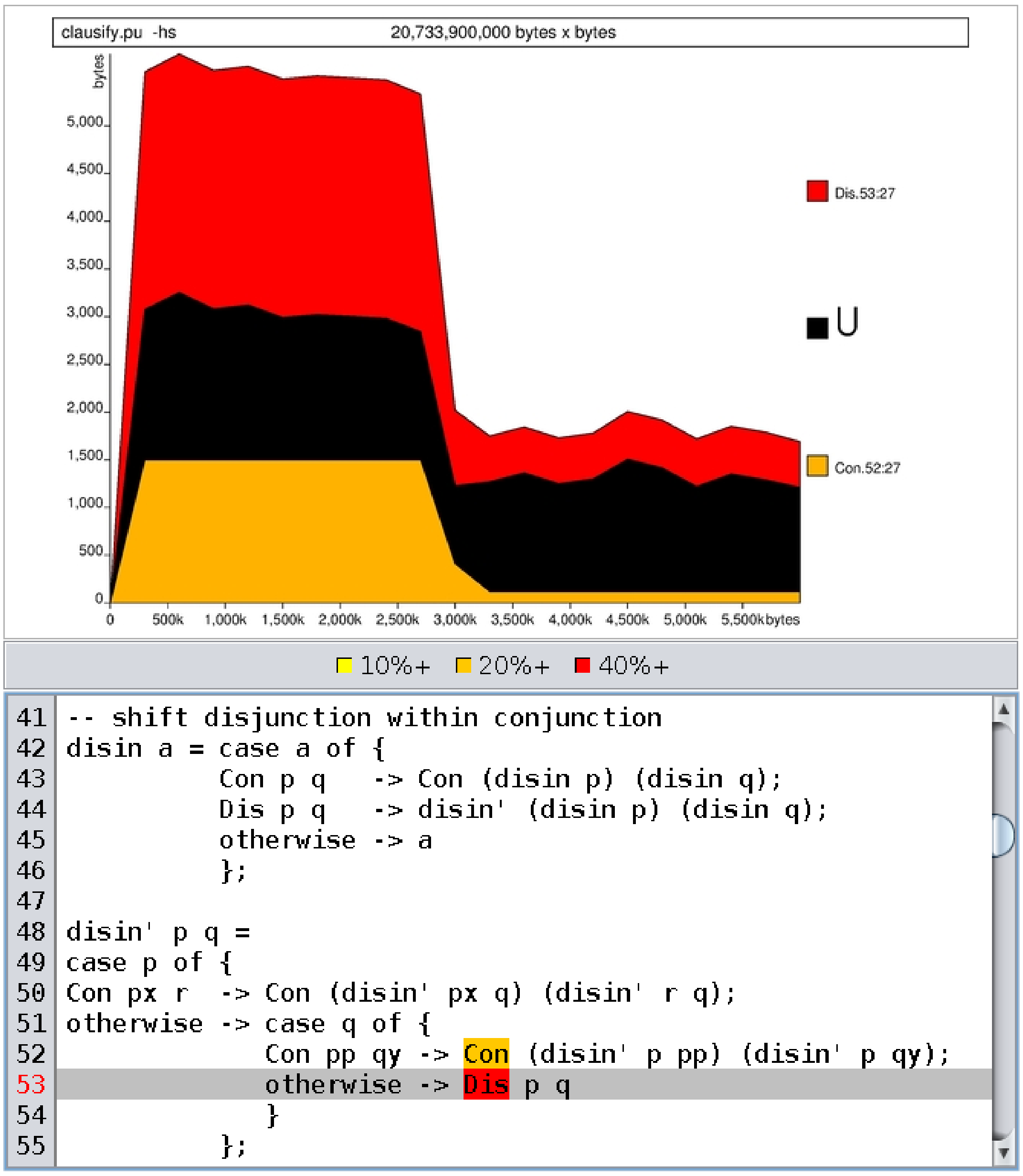
The revised version listed above is the last space improving reformulation of `disin` by Runciman and Wakeling [1]. Eventually, Runciman and Röjemo [17] diagnosed the space leak caused by `disin` using advanced heap profiling tools (e.g. *retainer profile*). They employed a program transformation technique termed *filter promotion* to prune the `Formula` tree at an earlier stage in the `clausify` pipeline by introducing the *normalising constructor* `dis`. The result is a reduction in space from 7 kb to 1 kb.

However, the further information provided by the hotspot profile (in Figure 6.7) might have allowed the space consumption of `clausify` to be reduced further at an earlier stage.



```
-- shift disjunction within conjunction
disin prop =
case prop of {
Dis p q -> case q of {
       Con qq r  -> Con (disin (Dis p qq)) (disin (Dis p r));
       otherwise -> case p of {
                    Con pp rr ->
                    Con (disin (Dis pp q)) (disin (Dis rr q));
                    otherwise ->
                    let dp = disin p;
                        dq = disin q
                    in if or (conjunct dp) (conjunct dq)
                       then disin (Dis dp dq)
                       else Dis dp dq
                    }
       };
Con pp qq -> Con (disin pp) (disin qq);
otherwise -> prop
};
```

Figure 6.7: A hotspot profile of `clausify` after introducing `filterset`.



```
41 -- shift disjunction within conjunction
42 disin a = case a of {
43             Con p q   -> Con (disin p) (disin q);
44             Dis p q   -> disin' (disin p) (disin q);
45             otherwise -> a
46             };
47
48 disin' p q =
49 case p of {
50 Con px r  -> Con (disin' px q) (disin' r q);
51 otherwise -> case q of {
52                 Con pp qy -> Con (disin' p pp) (disin' p qy);
53                 otherwise -> Dis p q
54                 }
55             };
```

Figure 6.8: A hotspot profile after improving `disin`.

6.4 Evaluation

There are two functions which contribute to the *major* space leaks of the `clausify` program: `unicl` (for its *tail-strict* property [1]) and `disin` (for its *path-strict* property [17]).

The stack producer profile reveals `unicl'` (an auxiliary of `unicl`) as a major producer of stack frames (Figure 6.4). None of the heap profiles for `clausify` in [1][7][17] showed `unicl'` as a key component.

The function `foldrUnicl`, whose application frames occupy most of the stack space, does also appear in the producer (Figure 6.1) and construction (Figure 6.2) profiles of the heap. However, its indirect influence on heap space demands is not obvious from these heap profiles alone.

The hotspot profile (Figure 6.7) revealed the hotspot occurrence `Dis`, which strongly suggested the path-strict behaviour of `disin`. The lack of any occurrence information in the original producer and construction profiles [1] led to uncertainty about the space consumption of `Dis` and `Con` constructions [17]. The suggestion of Runciman and Röjemo in [51] that *occurrence* profiling could help to diagnose space leaks is thus confirmed.

Chapter 7

Conclusion

This dissertation has described the design, implementation and use of two new variants of memory profiling tools for a lazy functional language: a *hotspot profiler* (in Chapter 4) and a *stack profiler* (in Chapter 5).

A hotspot profile presents information in two forms: profile charts and *hotspots* highlighted by source occurrence. The profile chart represents a *hotspot-construction* profile, distributed by hotspot *temperatures*. Hotspots are also marked in a textual display of source programs with the temperature they represent. Further information about hotspots is given in individual profiles.

A stack profile provides different views of stack memory including, a *producer* profile, a *construction* profile, and their combinations. A producer is the function that introduces a frame in stack memory, and a construction is the function application that a frame represents.

7.1 Results Summary

Applying the hotspot profiler to small programs (in Section 4.3) revealed hotspots of different temperatures. Investigating the information provided for individual hotspots (e.g. *lifetime profile*) allowed us to reason about space consumption at the occurrence level and so dramatically reduce the space associated with hotspots.

Applying the stack profiler to small programs (in Section 5.5) revealed the functions causing excessive stack demands. Examining these functions allowed us to dramatically reduce stack space.

The usefulness of a hotspot profile greatly depends on the memory behaviour of the programs. For programs with a very even distribution of space demands, the hotspot profile provides no information as occurrences associated with low temperature (e.g. less than 10% of memory) are collapsed to a union band.

Stack profiles draw the programmer's attention to the functions causing stack overflow in a simpler way than heap profiles, since stack contents are *closures* only.

Furthermore, stack profiles can shed light on space leaks that are not fully apparent in heap profiles. Likewise, heap profiles can shed light on space leaks that are not fully apparent in stack profiles. An example of this is the `queens` program in Section 5.5.2; the function `length`, whose frames occupied the majority of the stack, indirectly influenced heap demands, causing an accumulation of `Cons` cells. On the other hand, the accumulation of memory cells produced by the function `toOne` appeared as a minor section in the stack producer profile.

A general technique is to obtain both stack and heap profiles, since each profile may point to different sources of space leaks.

The hotspot and stack profilers were also applied to a larger program called `clausify` (in Section 6.3). This program was extensively profiled in the literature, leading to a heap memory reduction from a peak of 1.3 Mb to a peak of 1 kb [1][17]. Two major space leaks of `clausify` are caused by the functions `unicl` (for its *tail-strict* property) and `disin` (for its *path-strict* property).

The stack producer profile (Figure 6.4) revealed `unicl'` (an auxiliary of `unicl`) as a major producer of stack frames. None of the heap profiles in [1][7][17] revealed `unicl'` as a key element.

The hotspot profile (Figure 6.7) revealed the hotspot occurrences `Dis` and `Con`, which strongly suggested the path-strict behaviour of the function `disin`. This further information might have led to a further reduction of space consumption for `clausify` at an earlier stage in [1].

7.2 Future Work

Concerning the hotspot profiler, the simple scheme of profiling by individual occurrence may produce results that are too fine-grained for some programs. This can be improved by providing a coarser form of hotspot. One possibility is to aggregate the cost of *compound constructions*, such as:

```
(Cons x (Cons y (Cons z Nil)))
```

In our current hotspot profiler, each `Cons` is an individual occurrence which may fall below the threshold to be classified as a hotspot. However, the combined cost of the occurrences of `Cons` could be significant.

Furthermore, the current method of profiling hotspots may not be suitable for large programs with modules, since every occurrence is profiled individually. For very large programs, it could even be appropriate to extend hotspots to a *module* [24]. The module of interest could then itself be profiled using fine-grained hotspots.

Bands in the hotspot-construction profile can have the same colour if there are hotspots of the same temperature. A different representation of a hotspot construction would be more helpful to distinguish hotspot bands of the same temperature more easily.

The individual profiles of hotspots only provide a lifetime profile. Support for different kinds of profile may be useful in diagnosing some problems. For example, to identify the last problem of the `execute` program in Section 4.3.2, we had to use our knowledge of the implementation. A *retainer* profile [17] would have been helpful.

Concerning the stack profiler, the current construction profile cannot distinguish between the different instances of function applications. Furthermore, functions which reuse a stack frame to represent another function are not reflected in stack profiles. However, this was only a brief investigation and it could be taken further.

In Section 5.5.3 we mentioned that after stack memory demands were dramatically reduced for the `compile` program, heap pressure remained. Techniques for *stack and heap trade-offs* would have helped to move parts of the heap pressure to stack memory.

To understand the nature of the space leaks caused by the functions `length` (Section 5.5.2) and `unicl'` (Section 6.3), we had to *relate* the information provided by stack and heap profiles. So there is a gap between stack and heap profiles. A technique to link stack and heap profiles would have been helpful.

Extra information for stack frames might, perhaps, have improved experience with stack profiling, e.g. *lifetime* and *biographical* profiles [39] for stack frames.

Appendices

Appendix A

The G-Machine

This appendix contains the semantics of the G-machine employed for the *Core* language. We present the *compilation schemes* and *state transition rules* of the G-machine, respectively.

A.1 Compilation Rules

The abstract compiler that generates G-machine code is divided into four main compilation schemes given below. The objects appearing in the compilation rules can be deduced from ranged metavariables, where

ρ	denotes a mapping from variable names to their stack offset, *relative to the top of the stack* with the topmost stack pointer starting at offset zero.		
f	denotes a function.	x	denotes a variable.
n	denotes a current stack depth.	e	denotes an arbitrary expression.
i	denotes an integer literal.	C_l	denotes a constructor C labelled l.

A.1.1 Scheme $\mathcal{F}$ (Function Definition)

$\mathcal{F}[\![fun]\!]$ generates G-machine code for a function definition fun.

$$\mathcal{F}[\![f\ x_1\ \ldots\ x_{\mathrm{n}} = e]\!] \quad = \quad \mathcal{R}[\![e]\!]\ [x_1 \mapsto 0,\ \ldots\ , x_{\mathrm{n}} \mapsto n-1]\ n$$

where f is a function name with n arguments. $\mathcal{F}$ calls $\mathcal{R}$ to compile code for the right-hand side of a function definition, passing an environment mapping from arguments' names to stack offsets and the number of arguments n. When a function is *entered*, the xth argument is $n-1$ elements relative to the top of the stack.

A.1.2 Scheme $\mathcal{R}$ (Return Value)

$\mathcal{R}[\![e]\!]\ \rho\ n$ generates G-machine code that instantiates a right-hand side expression e in environment ρ for a function with arity n, computes the value of e and updates the root application node of an application with a result, then returns from a function and proceeds to unwind the result.

$$\mathcal{R}[\![e]\!]\ \rho\ n \quad = \quad \mathcal{E}[\![e]\!]\ \rho +\!\!+ [\texttt{UPDATE}\ n, \texttt{POP}\ n, \texttt{UNWIND}]$$

A.1.3 Scheme $\mathcal{E}$ (Evaluate)

$\mathcal{E}[\![e]\!]\ \rho$ compiles code that evaluates expression e in environment ρ to *weak head normal form*, leaving a pointer to it on top of the stack. This scheme *short-circuits* graph construction for expressions occurring in *strict context*.

$$
\begin{array}{lcl}
\mathcal{E}[\![i]\!]\ \rho & = & [\texttt{PUSHINT}\ i] \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{let}\ x_1 = e_1 ; \ldots ; x_\mathrm{n} = e_\mathrm{n}\ \texttt{in}\ e]\!]\ \rho} \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} \mathbin{+\!\!+} \\
 & & \mathcal{E}[\![e]\!]\ \rho' \mathbin{+\!\!+} [\texttt{SLIDE}\ n] \\
 & & \quad \{\text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_\mathrm{n} \mapsto 0]\} \\
\multicolumn{3}{l}{\mathcal{E}[\![\texttt{letrec}\ x_1 = e_1 ; \ldots ; x_\mathrm{n} = e_\mathrm{n}\ \texttt{in}\ e]\!]\ \rho} \\
 & = & [\texttt{ALLOC}\ n] \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![e_1]\!]\ \rho' \mathbin{+\!\!+} [\texttt{UPDATE}\ n-1] \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![e_n]\!]\ \rho' \mathbin{+\!\!+} [\texttt{UPDATE}\ 0] \mathbin{+\!\!+} \\
 & & \mathcal{E}[\![e]\!]\ \rho' \mathbin{+\!\!+} [\texttt{SLIDE}\ n] \\
 & & \quad \{\text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_\mathrm{n} \mapsto 0]\} \\
\mathcal{E}[\![e_0 + e_1]\!]\ \rho & = & \mathcal{E}[\![e_1]\!]\ \rho \mathbin{+\!\!+} \mathcal{E}[\![e_0]\!]\ \rho^{+1} \mathbin{+\!\!+} [\texttt{ADD}] \\
 & & \quad \{\text{similarly for arithmetic and comparasion expressions}\} \\
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\ \rho & = & \mathcal{E}[\![e]\!]\ \rho \mathbin{+\!\!+} [\texttt{CASEJUMP}\ \mathcal{D}[\![alts]\!]\ \rho] \\
\mathcal{E}[\![C_l\ e_1\ \ldots\ e_n]\!]\ \rho & = & \mathcal{C}[\![e_n]\!]\ \rho^{+0} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}[\![e_1]\!]\ \rho^{+(n-1)} \mathbin{+\!\!+} [\texttt{CON}\ l\ n] \\
\mathcal{E}[\![e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho \mathbin{+\!\!+} [\texttt{EVAL}]
\end{array}
$$

A.1.4 Scheme $\mathcal{C}$ (Construct Graph)

$\mathcal{C}[\![e]\!]\ \rho$ compiles code which constructs the graph of expression e in an environment ρ and leaves a pointer to the graph of e on top of the stack.

$$
\begin{array}{lcl}
\mathcal{C}[\![f]\!]\ \rho & = & [\texttt{PUSHFUN}\ f] \\
\mathcal{C}[\![x]\!]\ \rho & = & [\texttt{PUSH}\ (\rho\ x)] \\
\mathcal{C}[\![i]\!]\ \rho & = & [\texttt{PUSHINT}\ i] \\
\mathcal{C}[\![e_0\ e_1]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho \mathbin{+\!\!+} \mathcal{C}[\![e_0]\!]\ \rho^{+1} \mathbin{+\!\!+} [\texttt{MKAP}] \\
\multicolumn{3}{l}{\mathcal{C}[\![\texttt{let}\ x_1 = e_1 ; \ldots ; x_\mathrm{n} = e_\mathrm{n}\ \texttt{in}\ e]\!]\ \rho} \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)} \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![e]\!]\ \rho' \mathbin{+\!\!+} [\texttt{SLIDE}\ n] \\
 & & \quad \{\text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_\mathrm{n} \mapsto 0]\} \\
\multicolumn{3}{l}{\mathcal{C}[\![\texttt{letrec}\ x_1 = e_1 ; \ldots ; x_\mathrm{n} = e_\mathrm{n}\ \texttt{in}\ e]\!]\ \rho} \\
 & = & [\texttt{ALLOC}\ n] \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![e_1]\!]\ \rho' \mathbin{+\!\!+} [\texttt{UPDATE}\ n-1] \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![e_n]\!]\ \rho' \mathbin{+\!\!+} [\texttt{UPDATE}\ 0] \mathbin{+\!\!+} \\
 & & \mathcal{C}[\![e]\!]\ \rho' \mathbin{+\!\!+} [\texttt{SLIDE}\ n] \\
 & & \quad \{\text{where } \rho' = \rho^{+n}[x_1 \mapsto n-1, \ldots, x_\mathrm{n} \mapsto 0]\} \\
\mathcal{C}[\![C_l\ e_1\ \ldots\ e_n]\!]\ \rho & = & \mathcal{C}[\![e_n]\!]\ \rho^{+0} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathcal{C}[\![e_1]\!]\ \rho^{+(n-1)} \mathbin{+\!\!+} [\texttt{CON}\ l\ n]
\end{array}
$$

A.1.5 Miscellaneous schemes for `case` expressions

$\mathcal{D}[\![alts]\!]\ \rho$ compiles code for the alternative branches $alts$ within a `case` expression in environment ρ, using the auxiliary rule $\mathcal{A}$.

$$\mathcal{D}[\![alt_1\ \dots\ alt_n]\!]\ \rho \quad = \quad [\mathcal{A}[\![alt_1]\!]\ \rho\ , \dots,\ \mathcal{A}[\![alt_n]\!]\ \rho]$$

$\mathcal{A}[\![alt]\!]\ \rho$ compiles code for an alternative branch alt within a `case` expression.

$$\mathcal{A}[\![C_l\ x_1 \dots x_n\ \texttt{->}\ e]\!]\ \rho \quad = \quad l\ \texttt{->}\ [\texttt{SPLIT}\ n] \mathbin{+\!\!+} \mathcal{E}[\![e]\!]\ \rho' \mathbin{+\!\!+} [\texttt{SLIDE}\ n]$$
$$\{\text{where } \rho' = \rho^{+n}[x_1 \mapsto 0, \dots, x_n \mapsto (n-1)]\}$$

A.2 State Transition Rules

The G-machine is a stack-based *finite-state machine*. A *state* in the G-machine consists of a 6-tuple $\langle O, I, S, D, H, E \rangle$, where

O is the output produced to the standard channel.

I is the G-code stream under execution denoted by $c : i$, where c is the instruction currently being executed and i is the rest of the G-code to be executed.

S is a stack of pointers to heap node addresses. The notation $a : s$ stands for a stack whose top element is a, where s is a stack S.

D is the *dump* which is a stack of $\langle I, S \rangle$ pairs, where I is a G-code sequence and S is a stack. A dump D whose topmost pair is $\langle I, S \rangle$ is denoted by $\langle I, S \rangle : D$. The current stack S and the code sequence I are dumped in this component when EVAL is recursively called.

H is the heap (or *graph*) mapping heap node addresses to heap nodes. The notation $H[a \mapsto n]$ means the address a in the heap H points to node n. There are six types of heap node:

FUN n c	a function node, where n is the *arity* and c is the code sequence of a function;
APP a_1 a_2	an application node, where a_1 and a_2 are pointers to a function and its argument, respectively;
INT n	an integer node, where n denotes an integer;
IND a	an indirection node used to update the root of a *redex* or, in constructing *cyclic graphs* within *letrec* constructs, where a is pointer to a (shared) expression;
CONSTR l n	a constructor node, where l is a label identifying a constructor having n arguments.
HOLE	a hole node to be filled with an expression later in a computation.

E is the global environment, mapping function names to pairs containing the arity of a function and its code. The notation $E[f \mapsto n\ c]$ denotes a mapping from a function f to a pair of arity n and code c.

The operational semantics of the G-machine are described by means of *state transition rules*. Throughout, the notation [. . .] denotes a sequence whereas [ ] stands for an empty sequence.

A.2.1 Heap and stack operations

The stack is addressed by an offset with the top element on the stack having offset zero. Items below the top of the stack are indexed relative to the top of the stack. The `PUSH` n instruction takes an item at offset n in the stack and pushes this item onto the top of the stack. The `PUSHFUN` f instruction pushes onto the stack a pointer to a function node (FUN) containing arity n and code c of a function f from environment E. The instructions `PUSHINT` n and `CON` l n allocate an integer node (INT) and a constructor node (CONSTR) into the heap, respectively. The `MKAP` instruction allocates an application node APP a_1 a_2 into the heap for the two topmost items on the stack as operands a_1 and a_2 for the APP node, respectively.

$$\begin{array}{lrrllll}
\textbf{1.}\ \langle\, O, & \texttt{PUSH}\, n : i, & a_0 : .. : a_n : s, & D, & H, & E & \rangle \\
\Rightarrow \langle\, O, & i, & a_n : a_0 : .. : a_n : s, & D, & H, & E & \rangle \\
\textbf{2.}\ \langle\, O, & \texttt{PUSHFUN}\, f : i, & s, & D, & H, & E[f \mapsto n\ c] & \rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{FUN}\ n\ c], & E & \rangle \\
\textbf{3.}\ \langle\, O, & \texttt{PUSHINT}\, n : i, & s, & D, & H, & E & \rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{INT}\ n], & E & \rangle \\
\textbf{4.}\ \langle\, O, & \texttt{CON}\, l\ n : i, & a_1 : .. : a_n : s, & D, & H, & E & \rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ l\ [a_1, ..., a_n]\ ], & E & \rangle \\
\textbf{5.}\ \langle\, O, & \texttt{MKAP} : i, & a_1 : a_2 : s, & D, & H, & E & \rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{APP}\ a_1\ a_2], & E & \rangle
\end{array}$$

A.2.2 Evaluation instructions

The `EVAL` instruction evaluates the expression on top of the stack to a WHNF. If the expression pointed to by the top of the stack is already in WHNF then the instruction `EVAL` does nothing. Otherwise, `EVAL` allocates a stack frame for the evaluation of this expression, saving the old stack and the current instruction stream in the dump component D before entering the UNWIND state.

The purpose of the `UNWIND` instruction is to prepare the stack for the evaluation of an expression. The `UNWIND` transition rule that is executed depends on the element on top of the stack. If the top of the stack is an application node, execution proceeds by unwinding the spine of a graph, pushing pointers to the application nodes along the way. Having reached a function, the `UNWIND(11)` instruction *rearranges* the stack, leaving pointers to the arguments of the function for faster access. The `ENTER` instruction then *enters* the code for the function on top of the stack to execute it. There are two other cases to deal with. If the top of the stack is an indirection node then the `UNWIND` instruction simply replaces the item on top of the stack with the item pointed to by the indirection node. Otherwise, if the top of the stack item is in WHNF then the old instruction stream and stack are restored from the dump component D.

The effect of the `UPDATE` n instruction is to overwrite the root node of an expression (at n+1th stack offset) with an indirection node pointing to the new expression at the top of the stack. The `POP` n instruction tidies up the stack by removing n unneeded stack items.

$$\begin{array}{lllllll}
\textbf{6.} & \langle\, O, & \texttt{EVAL} : i, & a : s, & D, & H, & E\,\rangle \\
\Rightarrow & \langle\, O, & [\texttt{UNWIND}], & [a]\,, & \langle i, s\rangle : D, & H, & E\,\rangle \\
\textbf{7.} & \langle\, O, & [\texttt{UNWIND}], & a : s, & D, & H[a \mapsto \text{APP}\ a_1\ a_2], & E\,\rangle \\
\Rightarrow & \langle\, O, & [\texttt{UNWIND}], & a_1 : a : s, & D, & H, & E\,\rangle \\
\textbf{8.} & \langle\, O, & [\texttt{UNWIND}], & a_0 : s, & D, & H[a_0 \mapsto \text{IND}\ a], & E\,\rangle \\
\Rightarrow & \langle\, O, & [\texttt{UNWIND}], & a : s, & D, & H, & E\,\rangle \\
\textbf{9.} & \langle\, O, & [\texttt{UNWIND}], & a : s, & \langle i', s'\rangle : D, & H[a \mapsto \text{INT}\ n], & E\,\rangle \\
\Rightarrow & \langle\, O, & i', & a : s', & D, & H, & E\,\rangle \\
\textbf{10.} & \langle\, O, & [\texttt{UNWIND}], & a : s, & \langle i', s'\rangle : D, & H[a \mapsto \text{CONSTR}\ n\ as], & E\,\rangle \\
\Rightarrow & \langle\, O, & i', & a : s', & D, & H, & E\,\rangle \\
\textbf{11.} & \langle\, O, & [\texttt{UNWIND}], & a_0 : .. : a_n : s, & D, & H\left[\begin{array}{l} a_0 \mapsto \text{FUN}\ n\ c \\ a_1 \mapsto \text{APP}\ a_0\ a'_1 \\ \quad\ldots \\ a_n \mapsto \text{APP}\ a_{n\text{-}1}\ a'_n \end{array}\right], & E\,\rangle \\
\Rightarrow & \langle\, O, & [\texttt{ENTER}], & a_0 : a'_1 : .. : a'_n : a_n : s, & D, & H, & E\,\rangle \\
\textbf{12.} & \langle\, O, & [\texttt{ENTER}], & a_0 : a_1 : .. : a_n : s, & D, & H[a_0 \mapsto \text{FUN}\ n\ c], & E\,\rangle \\
\Rightarrow & \langle\, O, & c, & a_1 : .. : a_n : s, & D, & H, & E\,\rangle \\
\textbf{13.} & \langle\, O, & \texttt{POP}\ n : i, & a_1 : .. : a_n : s, & D, & H, & E\,\rangle \\
\Rightarrow & \langle\, O, & i, & s, & D, & H, & E\,\rangle \\
\textbf{14.} & \langle\, O, & \texttt{UPDATE}\ n : i, & a : a_0 : .. : a_n : s, & D, & H, & E\,\rangle \\
\Rightarrow & \langle\, O, & i, & a_0 : .. : a_n : s, & D, & H[a_n \mapsto \text{IND}\ a], & E\,\rangle
\end{array}$$

A.2.3 `case` and `let(rec)` instructions

The `CASEJUMP` instruction expects a constructor node on top of the stack. The label l of this constructor is used to select the code sequence of an alternative and then the current code stream is prefixed by the code sequence of the selected alternative. The code sequence of an alternative starts with a `SPLIT` n instruction and ends with a `SLIDE` n instruction where n is the number of constructor arguments. The former instruction provides access to constructor components by "unpacking" them onto the stack, whereas the latter instruction removes unneeded pointers from the stack. Similarly, the instruction `SPLIT` n is used to remove local bindings within *let*(*rec*) constructs.

$$\begin{array}{llllllll}
\textbf{15.} & \langle\, O, & \texttt{CASEJUMP}\ [.., l \mapsto i', ..] : i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ l\ ss], & E\,\rangle \\
\Rightarrow & \langle\, O, & i' : i, & a : s, & D, & H, & E\,\rangle \\
\textbf{16.} & \langle\, O, & \texttt{SPLIT}\ n : i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ l\ [a_1, .., a_n]\,], & E\,\rangle \\
\Rightarrow & \langle\, O, & i, & a_1 : .. : a_n : s, & D, & H, & E\,\rangle \\
\textbf{17.} & \langle\, O, & \texttt{SLIDE}\ n : i, & a_0 : .. : a_n : s, & D, & H, & E\,\rangle \\
\Rightarrow & \langle\, O, & i, & a_0 : s, & D, & H, & E\,\rangle \\
\textbf{18.} & \langle\, O, & \texttt{ALLOC}\ n : i, & s, & D, & H, & E\,\rangle \\
\Rightarrow & \langle\, O, & i, & a_1 : .. : a_n : s, & D, & H\left[\begin{array}{c} a_1 \mapsto \text{IND HOLE} \\ \ldots \\ a_n \mapsto \text{IND HOLE} \end{array}\right], & E\,\rangle
\end{array}$$

The instruction `ALLOC` n allocates n indirection nodes into the heap, with each pointing to a HOLE node to be overwritten with a recursive locally bound expression within a *letrect* construct, thereby constructing *cyclic graphs*.

A.2.4 Arithmetic and comparison instructions

Arithmetic and comparison instructions pop two topmost elements of the stack, *unbox* them and perform a specified primitive operation, then push the result on top of the stack *boxed* into a corresponding heap object. Comparison operations produce a nullary constructor of type `Bool` as a result (i.e. `True` or `False`).

$$\begin{array}{lllll}
\textbf{19.}\ \langle\, O, & \texttt{ADD} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{INT}\ (n_0 + n_1)], \quad E\,\rangle \\
\textbf{20.}\ \langle\, O, & \texttt{SUB} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{INT}\ (n_0 - n_1)], \quad E\,\rangle \\
\textbf{21.}\ \langle\, O, & \texttt{MUL} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{INT}\ (n_0 * n_1)], \quad E\,\rangle \\
\textbf{22.}\ \langle\, O, & \texttt{DIV} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{INT}\ (n_0\ /\ n_1)], \quad E\,\rangle \\
\textbf{23.}\ \langle\, O, & \texttt{EQ} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ (n_0 == n_1)\ [\,]\,], \quad E\,\rangle \\
\textbf{24.}\ \langle\, O, & \texttt{NE} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ (n_0\ /{=}\ n_1)\ [\,]\,], \quad E\,\rangle \\
\textbf{25.}\ \langle\, O, & \texttt{LT} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ (n_0 < n_1)\ [\,]\,], \quad E\,\rangle \\
\textbf{26.}\ \langle\, O, & \texttt{LE} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ (n_0 <= n_1)\ [\,]\,], \quad E\,\rangle \\
\textbf{27.}\ \langle\, O, & \texttt{GT} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ (n_0 > n_1)\ [\,]\,], \quad E\,\rangle \\
\textbf{28.}\ \langle\, O, & \texttt{GE} : i, & a_0 : a_1 : s, & D, & H[a_0 \mapsto \text{INT}\ n_0,\ a_1 \mapsto \text{INT}\ n_1], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ (n_0 >= n_1)\ [\,]\,], \quad E\,\rangle
\end{array}$$

A.2.5 Printing instruction

The instruction `PRINT` converts the representation of a result during the execution of a program into a corresponding character string, concatenated to the output components produced so far. Provided that the result is a constructor, a pretty-printer (although not shown here) is used for this constructor and its components are evaluated in order to be printed.

$$\begin{array}{lllll}
\textbf{29.}\ \langle\, O, & \texttt{PRINT} : i, & a : s, & D, & H[a \mapsto \text{INT}\ n], \quad E\,\rangle \\
\Rightarrow \langle\, O \mathbin{+\!\!+} [n], & i, & s, & D, & H, \quad E\,\rangle \\
\textbf{30.}\ \langle\, O, & \texttt{PRINT} : i, & a : s, & D, & H[a \mapsto \text{CONSTR}\ l\ [a_1, .., a_n]\,], \quad E\,\rangle \\
\Rightarrow \langle\, O, & i' \mathbin{+\!\!+} i, & a_1 : .. : a_n : s, & D, & H, \quad E\,\rangle \\
\end{array}$$

$$\{\text{where } i' = [\underbrace{\texttt{EVAL},\ \texttt{PRINT}, \ldots,\ \texttt{EVAL},\ \texttt{PRINT}}_{n}]\}$$

A.2.6 Initial and final G-machine state

The G-machine starts its execution by evaluating a special CAF `main` via the code:

```
main:
   PUSHFUN main
   EVAL
   PRINT
```

Execution terminates when the current instruction stream is empty, leaving a result on top of the stack.

Appendix B

ME2G

ME2G is a program that reads *me*mory profiling data (i.e. heap or stack) and transforms profiling information *to* a graphical form. This appendix contains a summary of the profiling flags along with the graphical display commands for a hotspot profile that are provided by the ME2G program.

ME2G flags

Usage: *me2g* `options` *file.me*

Profile Type:	
`-hs`	hotspot profile
`-c`	construction profile
`-p`	producer profile
`-c -p` *list*	construction profile restricted by a list of producers
`-p -c` *list*	producer profile restricted by a list of constructions
Profile Options:	
`-hu`	hide the `Union` band
`-t` *t1,t2,t3*	set hotspots temperatures where $t1 < t2 < t3$
`-p`	use the previous scale of a profile
`-h` *text*	add a text to the header of a heap chart
`-ol`	omit lifetime profile in individual hotspots profiles
`-help`	print hotspot profile usage information

ME2G commands (Hotspot profile)

Commands:	
`Enter`	open the selected hotspot individual profile
`s`	save the profile of the selected window to a file
`f`	move the cursor forward to the next hotspot
`b`	move the cursor backward to the previous hotspot
`e`	close an individual hotspot profile
`q`	quit the hotspot profile by closing all the windows
`c`*[0-9]+*	jump to a specified column
`l`*[0-9]+*	jump to a specified line
`h`	show hotspot profile usage information

Appendix C

The `clausify` Program

This appendix contains the source of `clausify`: the essence of `clausify` (version 0) by Runciman and Wakeling [1]. The Pure version of `clausify` excludes the propositional formulae parser and formatting clausal axioms. A brief description of this program is included in Section 6.2.

```
data Pair a b = Pair a b;

data Eq = EQ | LT | GT;

-- abstract syntax for propositional formulae
data Formula = Sym Int
             | Not Formula
             | Dis Formula Formula
             | Con Formula Formula
             | Imp Formula Formula
             | Eqv Formula Formula;

-- convert a list of propositions to clausal forms
clausify p = mapClauses p;

mapClauses Nil          = Nil;
mapClauses (Cons x xs) = Cons (clauses x) (mapClauses xs);

-- transform a propositional formula to clauses
clauses p = unicl (split (disin (negin (elim p))));

-- 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 p q) = Con (elim (Imp p q)) (elim (Imp q p));

-- 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 (Not (Sym s))   = Not (Sym s);
negin (Dis p q)       = Dis (negin p) (negin q);
negin (Con p q)       = Con (negin p) (negin q);
negin (Sym s)         = Sym s;

-- shift disjunction within conjunction
disin prop =
case prop of {
Dis p q -> case q of {
       Con qq r  -> Con (disin (Dis p qq)) (disin (Dis p r));
       otherwise -> case p of {
                    Con pp rr ->
                    Con (disin (Dis pp q)) (disin (Dis rr q));
                    otherwise ->
                    let dp = disin p;
                        dq = disin q
                    in if or (conjunct dp) (conjunct dq)
                       then disin (Dis dp dq)
                       else Dis dp dq
                    }
       };
Con pp qq -> Con (disin pp) (disin qq);
otherwise -> prop
};

-- test for conjunctive proposition
conjunct (Con p q) = True;
conjunct (Dis p q) = False;
conjunct (Not p)   = False;
conjunct (Sym s)   = False;

-- split conjunctive proposition into a list of conjuncts
split p = split' p Nil;

split' (Con p q) a = split' p (split' q a);
split' (Dis p q) a = Cons (Dis p q) a;
split' (Not p)   a = Cons (Not p) a;
split' (Sym s)   a = Cons (Sym s) a;

-- form set of unique non-tautolous
-- clauses given list of conjuncts
unicl a = foldrUnicl Nil a;

unicl' p x = let cp = clause p
             in
             if tautclause cp then x else insertPair cp x;

foldrUnicl z Nil         = z;
foldrUnicl z (Cons x xs) = unicl' x (foldrUnicl z xs);
```

```
-- separate positive and negative
-- literals, eliminating duplicates
clause p = clause' p (Pair Nil Nil);

clause' (Dis p q)     (Pair c a) =
  clause' p (clause' q (Pair c a));
clause' (Sym s)       (Pair c a) = Pair (insert s c) a;
clause' (Not (Sym s)) (Pair c a) = Pair c (insert s a);

-- does any symbol appear in both
-- consequent and antecedent of clause
tautclause (Pair c a) = tautclause' c a;

tautclause' Nil y  = False;
tautclause' (Cons x xs) y
  | (isElem x y)   = True
  | otherwise      = tautclause' xs y;

-- does the element occur in the list?
isElem y Nil          = False;
isElem y (Cons x xs) = if (y == x)
                        then True
                        else isElem y xs;

-- insertion of an item into an ordered list
insert x Nil   = Cons x Nil;
insert x (Cons y ys)
  | (x < y)    = Cons x (Cons y ys)
  | (x > y)    = Cons y (insert x ys)
  | otherwise  = Cons y ys;

-- insertion of a pair into an ordered list
insertPair x Nil   = Cons x Nil;
insertPair x (Cons y ys) =
  case (pairEq x y) of {
  LT -> Cons x (Cons y ys);
  GT -> Cons y (insertPair x ys);
  EQ -> Cons y ys
  };

-- pair equality
pairEq (Pair a b) (Pair x y) =
  case (listEq a x) of {
  GT -> GT;
  LT -> LT;
  EQ -> listEq b y
  };

-- list equality
listEq Nil          Nil = EQ;
listEq Nil (Cons x xs) = LT;
listEq (Cons x xs) Nil = GT;
listEq (Cons x xs) (Cons y ys)
 | (x > y)   = GT
 | (x < y)   = LT
 | otherwise = listEq xs ys;

-- logical disjunction
or False x = x;
or True  x = True;

-- denotation of a
a = 1;

-- propositional formulae representing:
-- (a = a = a) = (a = a = a) = (a = a = a)
prop = Cons (Eqv (Eqv (Sym a) (Eqv (Sym a) (Sym a)))
                 (Eqv (Eqv (Sym a) (Eqv (Sym a) (Sym a)))
                      (Eqv (Sym a) (Eqv (Sym a) (Sym a)))))
       Nil;

main = clausify prop
```

Bibliography

[1] C. Runciman and D. Wakeling, "Heap profiling of lazy functional programs," *Journal of Functional Programming*, vol. 3, pp. 217–245, 4 1993.

[2] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2007.

[3] sleepyMonad, "Good introductory text about ghc implementation?." Available: `http://stackoverflow.com/questions/6048194/good-introductory-text-about-ghc-implementation`, 2011. [Online; accessed April. 26, 2016].

[4] E. Z. Yang and D. Mazières, "Dynamic space limits for Haskell," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 588–598, 2014.

[5] R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice Hall International Ltd, 1988.

[6] C. Runciman and D. Wakeling, "Problems & proposals for time & space profiling of functional programs," in *Functional Programming, Glasgow 1990* (S. Jones, G. Hutton, and C. Holst, eds.), Workshops in Computing, pp. 237–245, Springer, 1991.

[7] P. Sansom, *Execution profiling for non-strict functional languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1994.

[8] S. A. Jarvis, *Profiling large-scale lazy functional programs*. PhD thesis, Durham University, 1996.

[9] H. Nilsson, *Declarative debugging for lazy functional languages*. PhD thesis, Linköping University, 1998.

[10] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[11] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

[12] M. L. Scott, *Programming language pragmatics*. Morgan Kaufmann, 2000.

[13] G. Sussman, H. Abelson, and J. Sussman, *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass, 1983.

[14] P. Wadler, "A critique of Abelson and Sussman or why calculating is better than scheming," *ACM SIGPLAN Notices*, vol. 22, no. 3, pp. 83–94, 1987.

[15] B. Zorn and P. Hilfinger, "A memory allocation profiler for C and Lisp programs," in *Proceedings of the Summer USENIX Conference*, pp. 223–237, 1988.

[16] N. Röjemo, *Garbage collection, and memory efficiency, in lazy functional languages*. PhD thesis, Chalmers University of Technology, 1995.

[17] C. Runciman and N. Röjemo, "New dimensions in heap profiling," *Journal of Functional Programming*, vol. 6, pp. 587–620, 7 1996.

[18] J. Sparud, *Tracing and debugging lazy functional computations*. PhD thesis, Chalmers University of Technology, 1999.

[19] R. Watson, *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, Southern Cross University, 1997.

[20] J. P. Taylor, *Presenting the lazy evaluation of functions*. PhD thesis, Queen Mary and Westfield College, 1996.

[21] A. W. Penney, *Augmenting Trace-based Functional Debugging*. PhD thesis, University of Bristol, 1999.

[22] B. J. Pope, *A declarative debugger for Haskell*. PhD thesis, The University of Melbourne, 2006.

[23] S. Marlow, J. Iborra, B. Pope, and A. Gill, "A lightweight interactive debugger for Haskell," in *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pp. 13–24, ACM, 2007.

[24] C. Runciman and D. Wakeling, "Heap profiling of a lazy functional compiler," in *Functional Programming, Glasgow 1992* (J. Launchbury and P. Sansom, eds.), Workshops in Computing, pp. 203–214, Springer, 1993.

[25] N. Mitchell and C. Runciman, "Superflo: Making Haskell faster," *Implementation and Application of Functional Languages*, p. 334, 2007.

[26] G. Mainland, R. Leshchinskiy, S. P. Jones, and S. Marlow, "Haskell beats C using generalized stream fusion." Available: `http://research.microsoft.com/en-us/um/people/-simonpj/papers/ndp/haskell-beats-C.pdf`, Unpublished, 2013.

[27] S. Peyton Jones, W. Partain, and A. Santos, "Let-floating: moving bindings to give faster programs," in *ACM SIGPLAN Notices*, vol. 31, pp. 1–12, ACM, 1996.

[28] C. Reinke, "-O introduces space leak." Available: `https://ghc.haskell.org/trac/ghc/ticket/917`, 2006. [Online; accessed Feb. 20, 2015].

[29] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the conference on Functional programming languages and computer architecture*, pp. 223–232, ACM, 1993.

[30] P. N. Benton, *Strictness analysis of lazy functional programs*. PhD thesis, University of Cambridge, Computer Laboratory, 1993.

[31] J. Gustavsson, *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages*. PhD thesis, Chalmers University of Technology, 2001.

[32] B. O'Sullivan *et al.*, *Real World Haskell*. O'Reilly, 2008.

[33] N. Mitchell, "Leaking space," *Queue*, vol. 11, pp. 10:10–10:23, Sept. 2013.

[34] R. Bird, *Thinking Functionally with Haskell*. Cambridge University Press, 2014.

[35] C. J. Sampson, "Experience report: Haskell in the "real world": Writing a commercial application in a lazy functional lanuage," *ACM Sigplan Notices*, vol. 44, no. 9, pp. 185–190, 2009.

[36] C. Runciman and N. Röjemo, "Heap profiling for space efficiency," in *Advanced Functional Programming* (J. Launchbury, E. Meijer, and T. Sheard, eds.), vol. 1129 of *Lecture Notes in Computer Science*, pp. 159–183, Springer, 1996.

[37] J. Tibell, "Results from the State of Haskell, 2011 Survey." Available: `http://blog.johantibell.com/2011/08/results-from-state-of-haskell-2011.html`, 2011. [Online; accessed Feb. 25, 2015].

[38] N. C. Brown and A. T. Sampson, "A Trip Down Memory Lane in Haskell." Available: `http://twistedsquare.com/Haskell-Experience.pdf`, University of Kent. Unpublished, 2009.

[39] N. Röjemo and C. Runciman, "Lag, drag, void and use — heap profiling and space-efficient compilation revisited," in *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pp. 34–41, ACM, 1996.

[40] R. Jones, "Tail recursion without space leaks," *Journal of Functional Programming*, vol. 2, no. 01, pp. 73–79, 1992.

[41] O. Waddell and J. M. Ashley, "Visualizing the performance of higher-order programs," *ACM SIGPLAN Notices*, vol. 33, no. 7, pp. 75–82, 1998.

[42] S. Thompson, "Higher-order + Polymorphic = Reusable." Computing Laboratory, University of Kent, Tech. Rep., May 1997.

[43] A. W. Appel, B. F. Duba, D. B. MacQueen, and A. P. Tolmach, *Profiling in the presence of optimization and garbage collection*. Technical Report CS-TR-197-88, Princeton University, Dept. Comp. Sci., Princeton, NJ, 1987.

[44] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of Haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 12–1, ACM, 2007.

[45] A. L. d. M. Santos, *Compilation by transformation in non-strict functional languages*. PhD thesis, The University of Glasgow, 1995.

[46] G. D. Ripley, R. E. Griswold, and D. R. Hanson, "Performance of storage management in an implementation of SNOBOL4," *Software Engineering, IEEE Transactions on*, no. 2, pp. 130–137, 1978.

[47] P. H. Hartel and A. H. Veen, "Statistics on graph reduction of SASL programs," *Software: Practice and Experience*, vol. 18, no. 3, pp. 239–253, 1988.

[48] J. Wild, H. Glaser, and P. Hartel, "Statistics on storage management in a lazy functional language implementation," in *Parallel and distributed processing '91*, pp. 73–87, 1991.

[49] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović, "Generating object lifetime traces with Merlin," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 3, pp. 476–516, 2006.

[50] L. Augustsson and T. Johnsson, "The Chalmers Lazy-ML Compiler," *The computer journal*, vol. 32, no. 2, pp. 127–141, 1989.

[51] C. Runciman and N. Röjemo, "Two-pass heap profiling: A matter of life and death," in *Implementation of Functional Languages* (W. Kluge, ed.), vol. 1268 of *Lecture Notes in Computer Science*, pp. 222–232, Springer, 1997.

[52] C. Clack, S. Clayman, and D. Parrott, "Lexical profiling: theory and practice," *Journal of Functional Programming*, vol. 5, pp. 225–277, 4 1995.

[53] N. Röjemo, "Highlights from nhc—a space-efficient Haskell compiler," in *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pp. 282–292, ACM, 1995.

[54] "Profiling Memory Usage." Available: `https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/prof-heap.html`. [Online; accessed Feb. 01, 2015].

[55] P. M. Sansom and S. L. Peyton Jones, "Profiling lazy functional programs," in *Functional Programming, Glasgow 1992*, pp. 227–239, Springer, 1993.

[56] P. M. Sansom, "Time profiling a lazy functional compiler," in *Functional Programming, Glasgow 1993*, pp. 252–264, Springer, 1994.

[57] P. M. Sansom and S. L. Peyton Jones, "Time and space profiling for non-strict, higher-order functional languages," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pp. 355–366, ACM, 1995.

[58] J. Launchbury, "A natural semantics for lazy evaluation," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 144–154, ACM, 1993.

[59] P. M. Sansom and S. L. Peyton Jones, "Formally based profiling for higher-order functional languages," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 2, pp. 334–385, 1997.

[60] S. L. Peyton Jones and A. M. Santos, "A transformation-based optimiser for Haskell," *Science of computer programming*, vol. 32, no. 1, pp. 3–47, 1998.

[61] S. L. Peyton Jones, "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of functional programming*, vol. 2, no. 02, pp. 127–202, 1992.

[62] S. Marlow and S. L. Peyton Jones, "The new GHC/Hugs runtime system," *URL http://research. microsoft. com/apps/pubs/default. aspx*, 1998.

[63] R. G. Morgan and S. A. Jarvis, "Profiling large-scale lazy functional programs," *Journal of Functional Programming*, vol. 8, no. 03, pp. 201–237, 1998.

[64] "hp2ps–heap profile to postscript." Available: `https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/hp2ps.html`. [Online; accessed Feb. 01, 2015].

[65] "ghc/Type.lhs." Available: `https://github.com/ghc/ghc/blob/master/libraries/integer-gmp/GHC/Integer/Type.lhs#L108`. [Online; accessed Feb. 15, 2015].

[66] G. Novark, E. D. Berger, and B. G. Zorn, "Efficiently and precisely locating memory leaks and bloat," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pp. 397–407, ACM, 2009.

[67] R. Shaham, E. K. Kolodner, and M. Sagiv, "Heap profiling for space-efficient java," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pp. 104–113, ACM, 2001.

[68] S. Nilsson, "Heapy: A memory profiler and debugger for Python," Master's thesis, Linköping University, Department of Computer and Information Science, 2006.

[69] N. Hallenberg, "A Region Profiler for a Standard ML compiler based on Region Inference," Student Project 96-5-7, Department of Computer Science, University of Copenhagen, 1996.

[70] S. L. Peyton Jones and D. R. Lester, *Implementing Functional Languages: a tutorial*. Café Press, 2004.

[71] T. Johnsson, *Compiling lazy functional languages*. PhD thesis, Chalmers University of Technology, 1987.

[72] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[73] T. Johnsson, *Efficient compilation of lazy evaluation*, vol. 19. ACM, 1984.

[74] M. Drautzburg, "How to diagnose Stack Overflow in haskell." Available: `http://stackoverflow.com/questions/17734281/how-to-diagnose-stack-overflow-in-haskell?rq=1`. [Online; accessed Mar. 13, 2015].

[75] P. Wadler, “How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages,” in *Functional Programming Languages and Computer Architecture*, pp. 113–128, Springer, 1985.

[76] W. Partain, *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992*, ch. The nofib Benchmark Suite of Haskell Programs, pp. 195–202. Springer, 1993.