

Fixing Some Space Leaks without a Garbage Collector

Jan Sparud

Department of Computer Sciences
Chalmers University of Technology
Göteborg, Sweden

E-mail: `sparud@cs.chalmers.se`

April 4, 1993

Abstract

This paper describes a method for eliminating a certain class of space leaks in lazy functional languages. A program that space leaks consumes more memory than would be expected. This may lead to longer execution time, or that the program unnecessarily runs out of memory. It has been known for a long time that functions returning tuples may give rise to space leaks. Hughes [Hug83] has shown that some programs *must* leak memory, if a sequential¹ evaluator is used. Several researchers have tried to solve this problem, with varying approaches, by introducing the necessary parallelism. Some of them modify the garbage collector, or use special operators to parallelize the programs explicitly. The approach used here is to use pattern bindings in definitions, which are available in most functional languages, to control the parallelism. No modification of the garbage collector or special operators are needed. The method has been implemented in the Chalmers LML/HBC compiler [AJ93, AJ89].

1 Introduction

Lazy functional languages have many pleasant features. The laziness property makes programs postpone evaluation of expressions until the value of them are required. This minimizes the number of reductions performed, but it is hard to reason about how much memory the programs will consume. If a function uses more memory than necessary, less memory is available for the the rest of the computation, causing garbage collection more often than otherwise. Some optimizations (in terms of reduction steps), may lead to longer execution time, due to extra space needed. So when trying to reduce the number of re-

ductions in a program, one should also take into account that space usage has a time cost.

A function that uses unexpectedly much memory is said to have a *space leak*. At least three different classes of space leaks have been identified [Wad87]. Two of them can (to some extent) be cured with different kinds of program analysis, such as strictness analysis. The third class concerns functions returning more than one result, e.g., in tuples, and is what the method proposed in this paper takes care of.

The organization of this paper is as follows. Section 2 discusses space leaks, and one example of what kind of space leaks the proposed method can fix is given. Section 3 describes the general method that solves these kinds of space leaks. Section 4 discusses implementation issues. Some optimizations (assuming a stack based graph-reducing implementation) that improve the performance of the method are also presented. A program which suffered from a serious space leak is discussed in section 5. Our method solved the problem. An example where no rewriting exists that solves the space leak is possible (without using parallel evaluation) is also given. In section 6 we compare this method of eliminating space leaks with other methods. The conclusions are presented in section 7.

2 A space leak

The reason for the leaky behavior of functions returning tuples (or, more generally, compound values of any type) is that references to the values in the tuple are indirected via the tuple itself. For example, assume that one part of a tuple is evaluated and results in a very long list. Assume further that the list is not used again. Normally, the garbage collector should be able to detect this and free the memory used by the list. But as long as there are unevaluated references to *any* part of the tuple, *no* part of the tuple can be garbage collected, even if it is never to be used again.

¹A sequential evaluator is one that only reduces one expression (and its subexpressions, if needed) until it has been reduced to normal form.

To give a concrete example, consider the **break** function below (we write code examples in a Haskell-like [Hud92] notation). This is a slightly changed version of the example in [Wad87].

```
break (x : xs) = if x == '↵' then
    ([ ], xs)
else
    (x : ys, zs)
  where (ys, zs) = break xs
```

(1)

where ‘↵’ is the newline character. The function takes a list of characters and returns a pair of lists, one containing all characters up to the first newline character, and the other containing all characters after it.

Now we use **break** in a program that reads input as a lazy list of characters and echoes it, with a line saying ‘surprise’ inserted after the first line. ‘++’ is the list concatenation operator.

```
first ++ "↵surprise↵" ++ rest
where (first, rest) = break input
```

(2)

Intuitively, this program will run in constant space. Every character read is examined and printed, except that the first newline character encountered is replaced with the surprise line. The fact is that the space used when running this program (assuming lazy sequential evaluation) is proportional to the length of the first line. The reason for this is that both **first** and **rest** refers to the pair returned by **break input**. When a character in **first** is written, it is no longer needed, and should thus be possible to garbage collect. But since **rest** is not yet evaluated, it still points to the tuple containing all of the input, and thus nothing can be garbage collected.

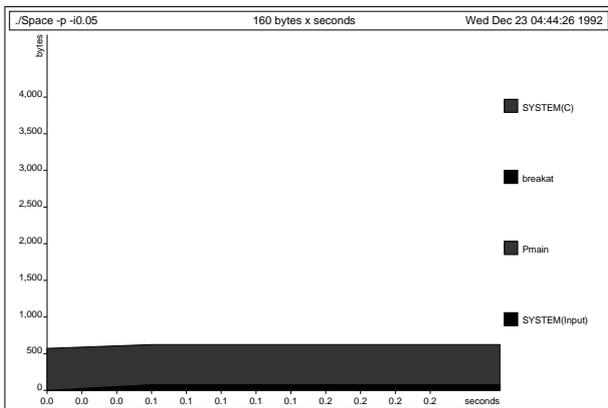


Figure 1: The heap usage when running the **surprise** program using the proposed method.

Since the first line can be arbitrarily long, it is possible that the program runs out of memory, when it, intuitively, should be running in constant space. This

example program is small and maybe not very useful, but as we will see in section 5, the tuple returning problem will have much worse implications.

The figures 1 and 2 show the memory usage when running this program with and without our method. The input file and the graph scale are the same in both cases. The graphs are produced by the heap profiler presented in [RW93], and show how the amount of storage occupied by a program graph varies over the time that the program takes to run. We can see (by the shape of the curve) that the memory used is constant over the running time with our method. Figure 2 shows that the program leaks memory without the method.

Hughes [Hug83] has shown that this behavior is inherent to lazy sequential languages. The subject of this paper is to present a way of introducing ‘minimal parallelism’ so that the first time a value from a tuple is demanded, all references via the tuple to other values will be short circuited.

To make this work the programmer must use pattern binding in definitions, when returning multiple results, instead of using projection functions. The pattern bindings tells the compiler implicitly which expressions to parallelize. The parallelism is captured by a (non-pure) function, which is transparent to the programmer. This solution is thus a very simple one. There is no need to clutter the program with synchronization operators, neither needs the garbage collector be modified.

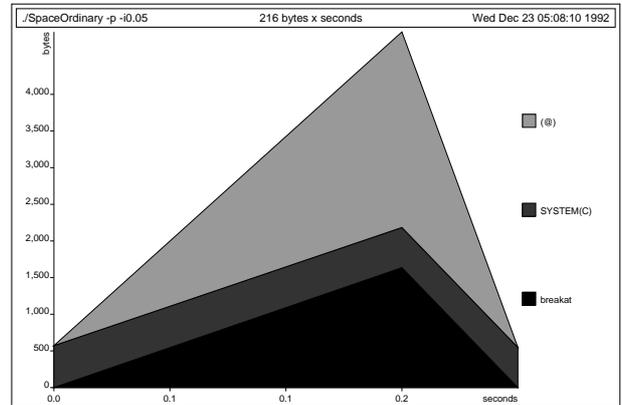


Figure 2: The heap usage when running the **surprise** program without using the proposed method.

3 The space leak fix

3.1 Pattern bindings

Pattern bindings (i.e., patterns on the left-hand side of a definition) in **where**-clauses and **let**-clauses, are allowed in many functional languages. As an example, see the **break** function (1) above.

The general form of a pattern binding is

$$P = E \quad (3)$$

where P is a pattern, binding at least one variable, and E is an arbitrary expression. The problem mentioned in section 2 occurs only when the pattern binds more than one variable.

This kind of pattern bindings are usually transformed to bindings that bind single variables only. This can be done in several ways (see section 3.2). When the expression E above is evaluated, we should check that the pattern specified by P really matches the result of E . This is called conformality check. Since pattern bindings are lazy (we are considering lazy languages), the conformality check should not be performed until the value of some of the variables in P is actually demanded. Pattern bindings will play a central part in our method.

3.2 Transformations

Peyton-Jones [PJ87] describes a method where the conformality check is performed once for each pattern. He suggests that (3) should be transformed into

$$\begin{aligned} e &= (\lambda P' \rightarrow (v'_1, v'_2, \dots, v'_n)) E \\ v_1 &= \text{sel}_{n1} e \\ v_2 &= \text{sel}_{n2} e \\ &\dots \\ v_n &= \text{sel}_{nn} e \end{aligned} \quad (4)$$

where v_i are the variables in P , $P' = P[v'_i/v_i]$, e is a new variable and sel_{nm} is a selector function that selects the m th component from an n -tuple. The common functions `fst` and `snd` correspond to sel_{21} and sel_{22} respectively. The variables in the pattern are packed into a tuple, which is constructed when the first of the variables is demanded. The variables are then extracted from the tuple with the selector functions when needed.

Augustsson [Aug87] describes the method that is used in the Chalmers LML compiler [AJ93, AJ89]. With this method, (3) will be transformed into

$$\begin{aligned} e &= E \\ v_1 &= \mathbf{case} \ e \ \mathbf{of} \\ &\quad P \rightarrow v_1 \\ v_2 &= \mathbf{case} \ e \ \mathbf{of} \\ &\quad P \rightarrow v_2 \\ &\dots \\ v_n &= \mathbf{case} \ e \ \mathbf{of} \\ &\quad P \rightarrow v_n \end{aligned} \quad (5)$$

where v_i are the variables in P and e is a new variable. This transformation has the disadvantage that the conformality checking is done once for each variable used, instead of once for each pattern, as in (4). On

the other hand, with (5) no tuple need to be built. Augustsson mentions that measurements have shown that, for efficiency, (5) is to be preferred for patterns with less than about five variables.

3.3 The new transformation

In the following, we start with Peyton-Jones' transformation (4).

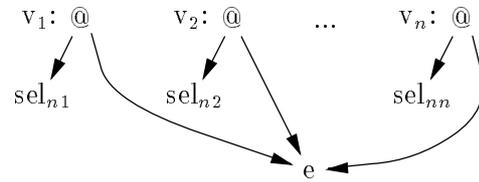


Figure 3: The graph of the pattern binding.

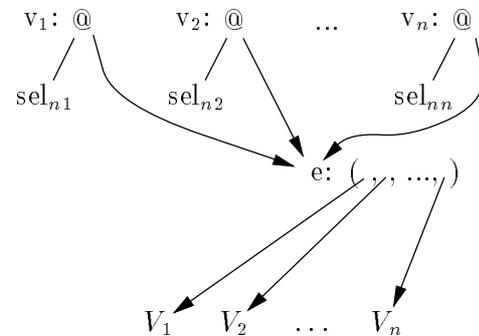


Figure 4: The graph after e has been evaluated.

Figure 3 shows the schematic graph of the definitions used in the transformation. '@' represents application nodes. The left branch of an application node is the function part, and the right branch the argument. As can be seen in figure 3, each variable bound in the pattern points to an expression with a selector function applied to an expression that, when evaluated, will be a tuple containing pointers to the value of every variable in the pattern. It is this that causes the leak possibility.

We note here that when e is evaluated to weak head normal form (i.e., in this case to tuple form), it is safe to apply the selector functions on all parts of the tuple, without losing laziness or risking non-termination (see figure 4).

The idea is to mimic the operation of the selector functions, by overwriting the variables with the corresponding parts of the tuple. But when does e get evaluated? It gets evaluated when the value of any variable bound in P is demanded for the first time. Then the value of e is demanded by one of the selector functions, E gets evaluated and matched against the pattern P (conformality check). If this fails, the

computation fails and the program terminates with an error message. If it succeeds, the tuple containing the variables of the pattern is created.

To know where to do the updating, e must have access to the pattern variables. Now we introduce a new (non-pure) function `updatePat`, that takes the variables and the created tuple as arguments, and returns the tuple.

We change the binding of e in (4) to

$$e = (\lambda P' \rightarrow \text{updatePat } (v'_1, v'_2, \dots, v'_n) \quad (v_1, v_2, \dots, v_n)) E \quad (6)$$

The type of `updatePat` is

$$\text{updatePat} :: \alpha \rightarrow \beta \rightarrow \alpha \quad (7)$$

As a side effect, `updatePat` overwrites every v_i with an indirection to the corresponding v'_i , thus all variables are updated in parallel. We must overwrite with indirections, since we otherwise risk to copy redexes.

This has the effect that the variables no longer have references to e but instead they point to their corresponding expressions, and the leak problem is solved.

Figure 5 shows the graph at this stage. ‘#’ represents indirection nodes. These will be short circuited by the garbage collector. Note that e is now unreferenced, and may be garbage collected.

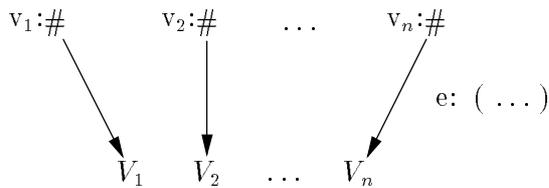


Figure 5: The graph when `updatePat` returns.

Note that the side effect is safe. It replaces expressions where a selector function is applied to a tuple (known to be in weak head normal form) with the part of the tuple that the selector function would have selected. It can be seen as a one step reduction on several expressions (one for each variable in the pattern) in parallel.

To summarize, the transformation proposed is

$$\begin{aligned}
 P &= E \\
 \Rightarrow \\
 e &= (\lambda P' \rightarrow \text{updatePat } (v'_1, v'_2, \dots, v'_n) \quad (v_1, v_2, \dots, v_n)) E \quad (8) \\
 v_1 &= \text{sel}_{n_1} e \\
 v_2 &= \text{sel}_{n_2} e \\
 \dots & \\
 v_n &= \text{sel}_{n_n} e
 \end{aligned}$$

This method has some disadvantages. One is that two tuples have to be built, instead of one. In the next section we will see how to avoid this.

4 Implementation

The method is applicable in all implementations of lazy functional languages that use graph reduction and can handle indirection nodes.

In this section we present some optimizations that can be performed, if the language implementation uses a stack machine, like the G-machine [Joh87].

`updatePat` is a function internal to the compiler, so the user can not use it explicitly. It is only used in pattern bindings, and the stack context is completely determined when it is applied. With that knowledge, we can improve the efficiency of the transformation above.

Instead of letting `updatePat` return the tuple created and then select the appropriate part with a selector function, we can let `updatePat` do all the work instead. When it is applied, we know the stack position of the pointer to the redex of the selector function that triggered the evaluation of `updatePat`. But `updatePat` overwrites that redex with an indirection to the expression that the selector function will select out of the tuple when `updatePat` returns. This means that if `updatePat` instead returns a pointer to that indirection, we do not need the selector function! We can replace it with a function that takes an argument, evaluates it and then returns the result. That is exactly what the identity function does!

This gives the final transformation:

$$\begin{aligned}
P &= E \\
\Rightarrow \\
e &= (\lambda P' \rightarrow \text{updatePat } (v'_1, v'_2, \dots, v'_n) \\
&\quad (v_1, v_2, \dots, v_n)) E \quad (9) \\
v_1 &= \text{id } e \\
v_2 &= \text{id } e \\
&\dots \\
v_n &= \text{id } e
\end{aligned}$$

where id is the identity function. This definition is not type correct, so if it is desirable to maintain type correctness throughout all transformations, one can use the last form in (8) as an intermediate representation, and then perform the last optimization when generating code. This also has the advantage that no code need to be generated for the selector functions.

In our implementation of this method in the Chalmers LML/Haskell compiler, the code generator recognizes applications of the selector functions, and replaces them with the identity function. It also recognizes applications of the `updatePat` function. `updatePat` always has two tuples as arguments: one with pointers to the variables in the pattern and one with expressions to overwrite the variables with. Instead of calling a function that does the work of `updatePat`, the code generator instantiates the body of `updatePat` wherever `updatePat` is used. This also means that the tuple containing the variables does not need to be built.

Even with these optimizations, there is an extra time cost with this method, in some situations. This means, that if a program uses pattern bindings in such a way that it does not leak memory without our method, it will run somewhat slower with our method.

5 A real example

This work was partly motivated by a very serious space leak problem in the implementation of a graphical user interface for lazy functional languages [CH93] called FUDGETS (functional widgets). FUDGETS consists to a large extent of functions manipulating streams (lazy lists). These functions are combined in different ways. If they are combined sequentially, no problems occur. But if they are combined so that two (or more) expressions use the same list, and the evaluation order forces one of the expressions to consume the list while the other still has a reference to the beginning of the list, it is possible that large amount of memory will be unnecessarily held on to, i.e., we

have a space leak.

In this case a major rewriting of the implementation was needed to avoid the space leak without our method. Since FUDGETS contains thousands of lines of code, this is not a small task. For an event driven program it is essential that the response time does not increase in proportion to the execution time, which could be the case if the program has a space leak, since garbage collecting would take longer and longer time. So if the program was not rewritten, some kind of parallel evaluation was needed. The minimal form of parallelism introduced by the method described in this paper was sufficient to solve the space leak in this case, and the only changes needed was to use pattern bindings in some critical functions.

The figures 6 and 7 show the memory usage when running a sample FUDGETS program (a simple mail reading program) with and without our method. The time shown on the x-axis does not include garbage collecting time.

As mentioned, our method can only solve space leaks with functions returning multiple results if pattern bindings are used. So it matters how the function definitions are written.

As an example, consider a function similar to the `break` function, the `partition` function. It takes a predicate and a list, and returns a pair of two lists, the first with all elements for which the predicate holds, and the second with all elements for which the predicate does not hold. The function can be defined in many ways, but one of the most natural is

$$\begin{aligned}
\text{partition} &:: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow ([\alpha], [\alpha]) \\
\text{partition } p \text{ } xs &= (\text{filter } p \text{ } xs, \text{filter } (\text{not } \cdot p) \text{ } xs)
\end{aligned} \quad (10)$$

where \cdot is an operator that composes two functions.

Unfortunately, this definition is leaky, even if the described method is used. If the expression that uses the result of `partition` consumes only one of the two parts of the tuple, while keeping a reference to the other, `xs` has to be evaluated. But the non-consumed part will point to the beginning of `xs`.

If we instead define `partition` as

$$\begin{aligned}
\text{partition} &:: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow ([\alpha], [\alpha]) \\
\text{partition } p \text{ } (x:xs) &= \\
&\quad \text{if } p \text{ } x \text{ then } (x:ts, fs) \text{ else } (ts, x:fs) \\
&\quad \text{where } (ts, fs) = \text{partition } p \text{ } xs
\end{aligned} \quad (11)$$

programs using it will not leak if the described method is used. The reason for this is that the list is only traversed once, and for every element both parts of the tuple returned will be updated.

We can draw some conclusions from this example. With the space leak fix method, seemingly equivalent definitions can have dramatically different space be-

havior. The two definitions of **partition** are both very reasonable, but the first (10) one is probably the most natural. It is of course a shortcoming of the method, that one can not write programs in the most natural way, without risking space leaks. We think, however, that the second definition (11) is not much less natural, and using pattern bindings in definitions where one think that there are the slightest possibility of a space leak is a cost that is acceptable.

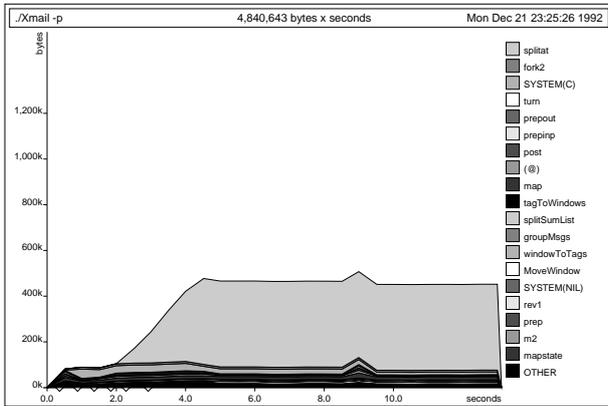


Figure 6: The heap usage when running the Xmail program using the proposed method.

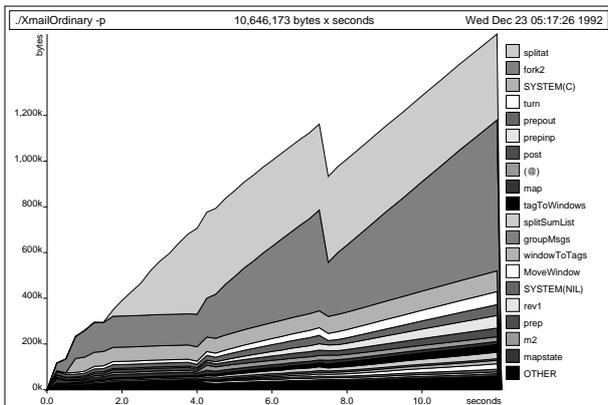


Figure 7: The heap usage when running the Xmail program without using the proposed method.

6 Related work

The problem of determining and controlling the space behavior of lazy functional programs have been addressed by several researchers. As mentioned, some programs are inherently leaky, with lazy sequential evaluation. To solve these problems, it is thus necessary to use some kind of parallelism. Hughes [Hug83] suggests using explicit synchronization and parallelism operators, to control the evaluation order in

cases where space leaks occur. This has some disadvantages. First, evaluation order annotations in programs make them harder to read. Second, the synchronization operator can, if improperly used, cause deadlock. Third, and most important, it is not always an easy task to see which expressions will have to be annotated to avoid space leaks.

Peyton-Jones [PJ87] discusses the run-time behavior of lazy functional programs in general. Some examples of semantically equivalent programs with dramatically different space usage in practice are given. Space leak problems with full laziness, as well as with accumulating functions are explained. He remarks that a programmer needs to know a lot about the particular language implementation, to be sure not to write leaky programs. In some cases, even a clever programmer cannot get around these problems, without language extensions. He mentions strictness analysis and parallel evaluation as examples. Our method is one such extension, that can solve the problem by using parallel execution. He also notes, that it is very difficult to tell where and why a program has a space leak, except that the program runs slower than expected. He asks for a set of debugging tools that can help the programmer in these cases.

One such tool, a heap profiler for the Chalmers LML/HBC compiler, is now available [RW93], and has been very helpful in the development of this work. The heap profiler can give dynamic information on the memory usage of a program. It looks in the heap at certain points of the execution and records how much live data there is and which functions have created what. The heap profiling information can then be viewed as a graph, which makes it possible to see if a program has a space leak, and if so, where the leak is located.

Wadler [Wad87] describes a method that uses the garbage collector to eliminate space leaks due to functions returning multiple results. This means that reduction rules must be defined for the garbage collector. The garbage collector recognizes applications of selector functions. If the argument is on tuple form, the selection is performed, otherwise the application is copied as normal. The garbage collector must contain such a rule for each selector function. The run-time cost is the checking that arguments are on tuple form. The only requirement our method has on the run-time system is that it has indirection nodes. This means that garbage collection can be separated from the evaluation.

7 Conclusions

The space leak fix method has been implemented in the Chalmers LML/HBC compiler. The basic idea is very simple and few parts of the compiler needed to be modified. As mentioned, it has been shown that with lazy sequential evaluation, some programs using functions that return multiple results, are inherently leaky. Our method captures the parallelism needed to cure this annoying space leak, by introducing a special function. This function uses a side effect to do the parallel work (i.e., simultaneously update several redexes), in a safe manner, so that the semantics of the language does not change. To tell the compiler which expressions to update simultaneously, the programmer must bind the expressions to variables in a pattern binding. This is a rather natural thing to do, when using functions returning multiple results. But if the programs are written in another way, our method does not fix the space leak. The overhead of the method, can make programs using pattern bindings (that normally will not leak) run somewhat slower.

8 Acknowledgements

I would like to thank my advisor Lennart Augustsson for always having time for questions and discussions. Magnus Carlsson and Thomas Hallgren gave me a large leaky program to test the method on. Staffan Truvé and Urban Boquist helped with commenting what eventually became this paper. Thanks also to John Hughes for comments and clarifications.

References

- [AJ89] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.
- [AJ93] L. Augustsson and T. Johnsson. *Lazy ML User's Manual*. Programming Methodology Group, Department of Computer Sciences, Chalmers, S-412 96 Göteborg, Sweden, 1993. Distributed with the LML compiler.
- [Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, November 1987.
- [CH93] M. Carlsson and T. Hallgren. FUDGETS - A Graphical User Interface in a Lazy Functional Language. To appear in Proc. FPCA 1993, March 1993.
- [Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.
- [Hug83] R. J. M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Programming Research Group, Oxford University, July 1983.
- [Joh87] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, February 1987.
- [PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
- [RW93] Colin Runciman and David Wakeling. Heap Profiling of Lazy Functional Programs. Technical report, Department of Computer Science, University of York, UK, to appear in Journal of Functional Programming, 3, 1993.
- [Wad87] P. Wadler. Fixing Some Space Leaks with a Garbage Collector. *Software Practice and Experience*, September 1987.