Tail recursion without space leaks

RICHARD JONES
Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent CT2 7NF, UK
(rej@ukc.ac.uk)

Abstract

The G-machine (Johnsson, 1987; Peyton Jones, 1987) is a compiled graph reduction machine for lazy functional languages. The G-machine compiler contains many optimizations to improve performance. One set of such optimizations is designed to improve the performance of tail recursive functions. Unfortunately, the abstract machine is subject to a space leak—objects are unnecessarily preserved by the garbage collector. This paper analyses why a particular form of space leak occurs in the G-machine, and presents some ideas for fixing this problem. This phenomena in other abstract machines is also examined briefly.

Capsule review

One of the disadvantages of functional programming is that small changes to a program can make rather large changes to its space behaviour. One form of this is called a *space leak*, where a large data structure is retained for a long time, even though it is eventually going to be discarded. This worsens performance by triggering garbage collection more often than would otherwise be required.

This paper describes a simple technique whereby a particular kind of space leak can be avoided, namely those rooted at graph nodes which are due to be updated. The idea is implemented in some functional-language compilers, but has not appeared in the open literature so far.

The problem is identified, the solution in the context of the G-machine is described, and the effectiveness of the proposed modification is measured.

1 Compilers for conventional imperative languages

How might a simple Pascal procedure, like the one shown below, be implemented?

procedure *f*;
begin ... *g*; **end**;

Typically, the procedure which called *f* would set up a new stack frame for it, including such information as return addresses and any arguments passed to *f*. In turn, a naïve implementation of *f* might set up a further stack frame for *g*. This is

https://doi.org/10.1017/S0956796800000277 Published online by Cambridge University Press

called a *tail call*. However, it is clearly unnecessary to retain *f*'s stack frame since its information (apart from the return address) will never be used again.

The implementation can be improved by *jumping* to *g* rather than *calling* it. In doing so, the same stack frame, including *f*'s return address, can be used for *g* as was used for *f*. If the tail is recursive, the situation is called *tail recursion*, and the effect of jumping rather than calling is to turn recursion into iteration which can be performed in constant space (Steele and Sussman, 1977; Bauer and Wössner, 1982).

2. Simple graph reducers get this optimization for free

Graph reduction (Wadsworth, 1971) is one method of implementing lazy functional languages. The essential idea behind it is as follows: to reduce an expression

(1) Construct the graph of the expression;
(2) Unwind the spine of this graph (leaving pointers to vertebrae—application nodes—on the stack) until the object in the function position (the leftmost, outermost term) is discovered;
(3) If this object is indeed a function, and all its arguments are present, apply the function to its arguments;
(4) Overwrite the redex node with the (root of the) result—it may in turn be a graph;
(5) Go to (2).

Turner (1979) observed that one property of graph reduction was to perform tail recursion in constant stack space *without* any explicit tail-call optimization being used. Nevertheless, the Construct–Unwind–Overwrite cycle wastes both time and heap space. For example, regardless of whether or not it is known at compile-time just what will be found in the leftmost, outermost position, the graph of an expression is still constructed, and its spine unwound. Such problems are addressed by compiled versions of graph reduction, which are discussed next.

3. The G-machine shortcircuits this mechanism

The G-machine is a very fast implementation of graph reduction based on compiling supercombinators (Hughes, 1984; Johnsson, 1987), that includes a large number of compile-time optimizations to improve performance. For tail calls in particular, rather than constructing the graph of functions applied to arguments, calling functions rearrange the Stack, replacing their own arguments with the arguments of the called function. The machine then *jumps* to the code of the called function. This is known as DISPATCHing.[1] Both calling and called function now share the same redex cell in the heap. If insufficient arguments are available, the graph of the application may have to be constructed and the redex overwritten in the same way as in the naïve reduction scheme.[2]

[1] The notation and method of Peyton Jones (1987) are used here.
[2] Actually, it is possible to do better than this. This scheme permits many other compile-time and run-time optimizations when the arity of the called function and the number of ribs currently on the Stack are known. Details are given in Peyton Jones (1987, Chapter 21).

https://doi.org/10.1017/S0956796800000277 Published online by Cambridge University Press

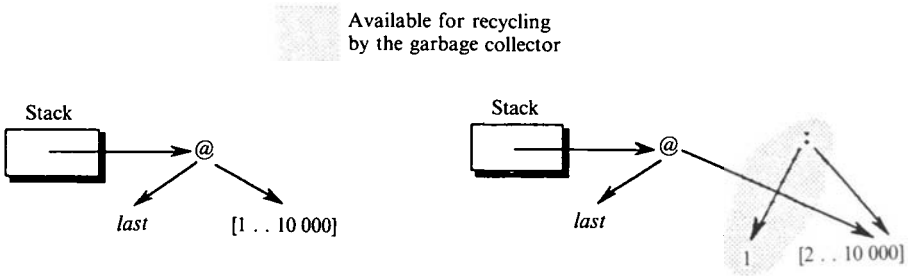

Fig. 1. After *UPDATE* at the end of each call to *last* in the naïve graph reducer.

4 A problem with tail recursion in the standard G-machine

Unfortunately, this scheme introduced a *space leak* (Wadler, 1987) not present in the naïve scheme, even though it uses less heap space. A space leak causes memory space to leak away invisibly; it occurs when a program retains a connection to an unnecessarily large graph, rather than releasing it for recycling by the garbage collector. For this reason, this phenomenon is also often known as *dragging*. Space leaks are a common problem in lazy functional languages (Hughes, 1984; Stoye, 1985; Peyton Jones, 1987, Chapter 23).

As a simple example, consider the function *last* which returns the last item of a (finite) list. It is assumed that the redex *does* contain the application, *last* [*1 . . 1000*]. This cell is the source of the space leak

```
let
  last :: [*] → *
  last l = if (t = nil)
           then (head l)
           else last t
           where t = tail l
in last [1 . . 1000]
```

Figure 1 shows the state of the naïve graph reducer after each call of *last*. At each step, the redex is overwritten with the application of *last* to the rest of the list. The front of the list is thus available for recycling by the garbage collector.

Notice that under lazy evaluation the list [*1 . . 10000*] might only be produced as *last* consumes it—the list need not exist at the beginning of the application *last* [*1 . . 10000*]. Thus space complexity can become constant rather than linear.

The G-machine, on the other hand, avoids overwriting the redex at each stage. Rather, the stack is rearranged to point at the tail of the list, and *last* is *DISPATCH*ed (Fig. 2) again. *No overwrite occurs until the end of the final tail call.* Consequently, the redex still holds pointers to the original argument, [*1 . . 1000*], no part of which can be reclaimed by the garbage collector.

https://doi.org/10.1017/S0956796800000277 Published online by Cambridge University Press

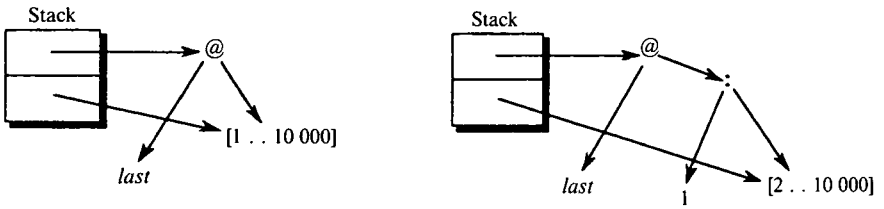

Fig. 2. Before *DISPATCH* at the end of each call to *last* in the standard G-machine.

5 Stop the garbage collector scavenging the redex

The solution is to prevent the garbage collector from reaching the arguments of the function via the redex node, and hence, marking them as wanted. Although the redex cell itself must be preserved by the garbage collector—it will be needed for overwriting by the eventual update (assuming termination)—the *contents* of this cell need not be preserved. The simplest way of doing this is to 'blackhole' the redex by overwriting (the tag of) the redex cell with a *HOLE* (Fig. 3).

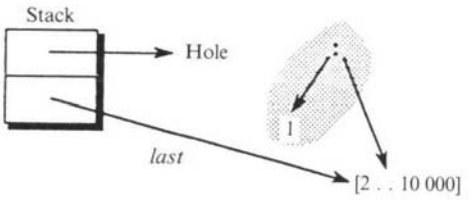

Fig. 3. 'Blackholing' the redex.

This is sufficient to remove the space leak as only the new argument(s) plus a single cell are preserved by the garbage collector. Most importantly, blackholing is safe. The redex is still the same cell in the heap, so sharing is preserved. The only way in which this scheme might break down is if the *contents* of the redex (rather than its address) were required before its reduction to WHNF were complete. In a sequential evaluator, this would mean that the value of the result of the reduction depended on itself! In this case, the program would fail to terminate in any case. A second advantage of blackholing is revealed: instead of falling into it, a black hole has been *actively* discovered.[3]

In a *parallel* evaluator, a second task might attempt to examine this *HOLE* cell before the first task updates it. One solution is to require the task which blackholes the redex to 'sign' it. If a task discovers one of its own blackholes then it should report non-termination. Any task discovering a blackhole which it does not 'own' should suspend itself until the owner of the blackhole overwrites this redex with a result. Tasks waiting on the blackhole should then be awakened. However, this scheme does not guarantee to detect self-dependence—for example, two mutually

[3] Similar cycles are also discovered by implementations that use *pointer reversal* rather than stacks to unwind the graph (Stoye *et al.*, 1984).

https://doi.org/10.1017/S0956796800000277 Published online by Cambridge University Press

dependent tasks, started simultaneously, would block on each other rather than reporting a black hole.

6 A modification to the standard abstract machine

There is no point in blackholing a redex that has already been blackholed. Ideally a cell should be blackholed just once, before the first function call that has this cell as its redex. Such new redexes can only be formed by unwinding the spine of an expression and discovering a function cell and sufficient arguments available. Assuming that the root node, the result of the program, is either a *HOLE* (or a special function, *program*, of arity 0, with no arguments to drag), this is the only case in which the redex should be blackholed on entry to the function.[4]

It is not necessary for all functions to blackhole a new redex *in order to avoid space leaks*. Any function that is guaranteed not to cause a heap space to be consumed, either directly or indirectly, cannot cause the garbage collector to be invoked, and therefore cannot cause a space leak. Examples might include functions returning a constant, projection functions, *head*, etc., depending on implementation. Furthermore, there is no point in blackholing the redex if the result of a function is certain to be a graph that contains references to *all* the arguments of the function. However, omitting black holes will mean that *non-termination due to self-dependence* may fail to be detected.

7 Other abstrct machines

The Spineless G-machine (Burn *et al.*, 1988) goes to some length to avoid creating spines. Without a spine, space leaks of the form considered here are not possible. The Spineless G-machine uses two different sorts of application node: the *SAP* for those applications which are both shared and reducible, and the *AP* for all other applications.

If the redex is an *AP*, unwinding will not cause a space leak as the pointer to the application node is removed from the stack. This node can now be recycled by the garbage collector (Fig. 4).

Fig. 4. Unwinding an unshared application cell.

On the other hand, if the redex is shared (a *SAP* cell), such a space leak is once again possible as the redex is still scavenged by the garbage collector. Blackholing plugs this lead in the Spineless G-machine (Fig. 5).

[4] Note that this 'new' redex might have been identified as a redex, and therefore blackholed earlier. However, this cannot in general be determined at compile-time, and it would be unnecessarily expensive to test for it at run-time.

https://doi.org/10.1017/S0956796800000277 Published online by Cambridge University Press

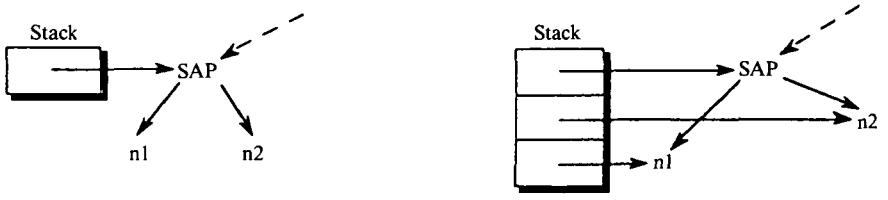

Fig. 5. Unwinding a shared application cell.

The TIM (Fairbairn and Wray, 1987) and the Spineless Tagless G-machine (Peyton Jones and Salkild, 1989) are compiled *closure* reducers rather than graph reducers. Nevertheless, the presence of TIM's 'markers' on the stack ('update frames' in the Spineless Tagless G-machine) to support laziness leads to the same problem—closures in the heap, that will not be used again until they are overwritten with a result, are still scavenged by the garbage collector. The solution is to overwrite the code pointer of every shared closure, on entry, with pointer for the 'blackhole' code. This prevents the garbage collector discovering other pointers in the closure, and again allows detection of certain forms of non-termination.

8 Results

Some results are presented in Table 1. In each case, the heap size of the machine was tuned to find the smallest possible heap in which the program could still run. Further tests reveal that with blackholing *last* now has constant residency rather than $O(n)$;

Table 1

Program	Minimum heap space requirement (words)	
	Without blackholing	With blackholing
last 1000	5211	223
reverse 2000	28252	22245
tree_sort 200	6481	4475
insert_sort 200	6063	4457
nth_prime 100	3689	2903
queens 8	996	837
digits of e 100	8555	7689

reverse (accumulating parameter version) has reduced its asymptotic residency by 21%, *tree_sort* by 32%, and *insert_sort* by 28%.

9 Conclusion

Space leaks are a major unresolved problem of lazy functional programming. It seems clear that there is no single overall solution to this problem, although a number of partial solutions have been proposed, e.g. (Wadler, 1987; Peyton Jones, 1987). In

https://doi.org/10.1017/S0956796800000277 Published online by Cambridge University Press

general, there are a number of distinct causes of space leaks, each requiring a different solution. The whole area requires careful attention in a functional language implementation.

In this article, attention has been drawn to a source of space leaks associated with tail call optimization, and it has been shown how this source can be removed. Moreover, a source of non-termination can now be *actively* discovered. The method is cheap to implement, safe, and plugs all space leaks of this form. Furthermore, it is minimal in the sense that it avoids performing multiple blackholing operations on known redexes, although it is possible that a redex might be blackholed more than once in the unwinding process. By statically analysing the code, further improvements can be made which specialize the actions of functions so that only those functions which could cause a space leak need to take action to avoid the leak. Finally, the application of this idea to other abstract machines has been sketched.

Acknowledgements

I am most grateful to David Turner, and to the anonymous referees for their useful comments on an earlier draft of this article, in particular for the mechanism used by the Spineless Tagless G-machine.

References

Bauer, F. L. and Wössner, H. 1982. *Algorithmic Language and Program Development*. Springer-Verlag, Berlin.

Burn, G. L., Peyton Jones, S. L. and Robson, J. D. 1988. The Spineless G-machine. In *ACM Conference on Lisp and Functional Programming*, pp. 244–258, Snowbird (July), USA.

Fairbairn, J. and Wray, S. C. 1987. Tim—a simple lazy abstract machine to execute supercombinators. In *IFIP Conference on Functional Programming and Computer Architecture*, pp. 34–45, Portland, USA (September).

Hughes, R. J. M. 1984. *The design and implementation of programming languages*, PhD thesis, PRG-40, University of Oxford, UK.

Johnsson, T. 1987. *Compiling Lazy Functional Languages*, PhD thesis, Chalmers University of Technology, Sweden.

Peyton Jones, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall, New York.

Peyton Jones, S. L. and Salkild, J. 1989. The Spineless Tagless G-machine. In *IFIP Conference on Functional Programming and Computer Architecture*, pp. 184–201, London, UK (August).

Steele, G. L. and Sussman, G. J. 1977. Lambda—the ultimate goto. Tech. Report 443, AI Memo, MIT AI Laboratory.

Stoye, W. R., Clarke, T. J. W. and Norman, A. C. 1984. Some practical methods for rapid combinator reduction. In *ACM Conference on Lisp and Functional Programming*, Austin, USA.

Stoye, W. R. 1985. *The implementation of functional languages using custom hardware*. PhD Thesis, Computing Laboratory, University of Cambridge, Tech. Report 81 (December).

Turner, D. A. 1979. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9: 31–49.

Wadler, P. 1987. Plumbers and dustmen: Fixing a space leak with a garbage collector. *Software—Practice and Experience*, 17: 595–608.

Wadsworth, C. P. 1971. *The Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, University of Oxford, UK.