Garbage collection, and memory efficiency, in lazy functional languages

Niklas Röjemo

May 31, 1995

Introduction

> "These introductions normally start with how great *lazy* functional languages are, how their mathematical purity makes proving the program correct easier, and how easy it is to code up difficult algorithms in a simple manner. They rarely talk about *memory management*. This is because this is a problem area for *lazy* functional languages."

The above is a paraphrase of [Sin92].[1] Although originally said in a slightly different context, it is also true as written here. One of the reasons functional languages are so nice is that the tedious memory management is handled automatically. Automatic memory management is an old idea [McC60], and is not unique for functional programs. It also exists for object oriented, logical and imperative programming languages. They also share the problem: automatic memory management takes time and space. In some cases, normally when the program uses complicated data structures, e.g., syntax trees, variable size lists, etc., nobody cares. Allocating and freeing memory "by hand" is so cumbersome and error prone in these cases, that most people accept the costs of automatic memory management. However, there are programs where the algorithm does not need a heap, but the language implementation does. In these cases it is understandable if programmers complain over the slow and memory hungry programs the compilers produce from their neat algorithms. This problem is clearly present in lazy functional languages, were lots of heap space is needed for normal execution.

The heap space is used for closures that represent suspended evaluations. These closures are often allocated at a tremendous rate, sometimes the limit is the bandwidth to the memory. The heap space needed for evaluation is in many

[1] The original text contained *interaction* instead of *memory management*, and did not specify *lazy*.

cases much larger than what an inexperienced programmer would expect. It is in fact often very difficult, even for an experienced programmer, to estimate the heap space needed in a lazy functional program.

This thesis presents five papers that aim at reducing the problem with memory management for lazy functional languages. The first two are improvements of garbage collectors for lazy functional languages; one for a parallel machine and one for a sequential machine. The third is an extended version of the heap profiling tools described by Runciman and Wakeling in [RW93b, RW93a]. The final two papers contain examples of memory efficient lazy functional programs; a set of parsing combinators and a Haskell compiler.

The following two sections give brief introductions to garbage collection and heap profiling respectively. After these a section about how to write space efficient functional programs follows. The introduction then concludes with the main contributions in the five presented papers.

Garbage collection – recycling memory

Garbage collection, sometimes called automatic memory management, is an old subject, and many surveys already exist, e.g., [Coh81, Wil92]. This section will therefore not be a survey of the field. Instead it concentrates on garbage collection for lazy functional languages, in particular how generational garbage collectors [LH83, Ung84] and lazy functional languages mix.

Generational garbage collection works best when the following three properties are fulfilled:

1. **Most nodes die young** [LH83, Ung84]. Concentrating collections on the young nodes therefore increases the amount of reclaimed memory for a given amount of work. It is only when not enough memory can be reclaimed by garbage collecting young nodes (*minor collections*) that the old nodes are collected (*major collections*). The intention is that there are many cheap minor collections between the major more expensive ones.

2. **Updates are rare.** Generational garbage collection takes advantage of the fact that almost all pointers point from younger nodes to older ones. The problem with updates is that they can create pointers from old nodes to young ones. All pointers from live old nodes must be followed during minor collection, otherwise young nodes, that are live, might be treated as dead. It is too costly to traverse the whole program graph for every minor collection to find these pointers. Most generational collectors therefore check all updates and remember the addresses of nodes, in the old heap, that are updated. Many updates can slow down the evaluation more than the speed-up achieved by using a generational collector.

3. **A small over-estimate of the live nodes does not lead to large loss of memory.** [WM89]. It is expensive to find exactly which nodes in the old heap that are live during a minor collection but a conservative approximation is easier: treat *all* nodes in the old heap as live during minor collection. This means that some old nodes that are dead might retain young nodes that ought to be reclaimed. If one old node often retains a lot of young nodes this can be a big problem.

However, lazy functional languages only fulfill the first one, they create a lot of short lived closures representing suspended evaluations. The other two points are however *not* good descriptions of lazy functional programs. Functional programs update a lot, at least if they are implemented with graph reduction [PJ87], and a small over-estimate of live nodes can prevent garbage collection of large data structures. However not all is lost despite this. Both the garbage collectors described in this thesis use the property that only some types of nodes can be updated in a lazy functional language.

The first approach, in "*A concurrent garbage collector for the* $\langle \nu, G \rangle$*-machine*", is never to consider updateable nodes as old. They are kept in the young generation until they are either updated, or they become garbage. This ensures that no updates can occur in the old heap, it is therefore not necessary to check updates at all. A problem is that some updateable nodes can get very old, and these nodes will be traversed during every minor collection.

Moving also updateable nodes into the old heap is therefore useful, but the check should be avoided if possible. And it is possible! How is described in "*Generational garbage collection, for lazy functional languages, without temporary space leaks*".

An over-estimate of live nodes is dangerous in lazy programming languages due to the use of large, sometimes potentially infinite, data structures. Only small parts of these structures are needed by the program at any time, and these parts are created on demand. If a node in one of these structures is wrongly considered live then no nodes reachable for this node can be reclaimed during garbage collections. The data structure will therefore use increasingly amount of memory as it is lazily created, instead of the small constant amount needed by the program. This problem is not as severe in strict languages due to the different programming style enforced by these languages.

The only solution so far is to be careful when moving nodes into the old heap, i.e., only nodes that has survived two, or maybe three, minor collections are moved into the old heap. This works very well since every specific part of the structure is only needed for a short time, and is therefore never moved into the old heap. However note that some lazy programs work well even when all nodes that survives one collection are moved into the old heap [App89, SJ93, Sew92]. A more in depth discussing of these problems are in "*Generational garbage collection, for lazy functional languages, without temporary space leaks*"

Heap profiling – do not create the garbage

Profiling has for a long time been an essential tool to develop efficient programs. Most compilers for imperative languages contain support for a time profiler, e.g., prof or gprof. There also exist tools to monitor memory management, e.g., purify [Inc94]. Purify monitors allocation and deallocation of memory, reporting deallocations of objects that are still used, or dead objects that are not deallocated. This tool is most often used to locate errors in the program. Time profiling is used to improve working programs, i.e., make them go faster. One could therefore think that languages with automatic memory management, and hence no errors connected with memory management, only need time profiles. This is however not true. Automatic memory management makes it difficult for the programmer to predict the memory needs for programs. Both allocation rate and residence can be much larger than expected. This is especially true for lazy functional languages where it is not always obvious when things are evaluated. Memory profiles are therefore needed to improve functional languages, i.e., reduce their memory consumption, memory residence and/or make them go faster.

Specialised versions of interpreters, and compilers, have been used by implementors of functional languages to determine heap behaviour in functional programs for some time; an example is Hartel and Veen [HV88]. Tools for the end users have however only appeared recently. The first widely used tools to observe memory residence are the heap profiles invented by Runciman and Wakeling [RW93b, RW93a]. These tools make it possible to observe *static* attribute of nodes in the heap such as *who* created the node, and *what* kind of nodes fill-up the heap. These tools have been extended by Runciman and myself also to include two *dynamic* attributes; *why* are nodes retained in the heap, and the *lifetime* of nodes. These extensions are presented in detail in the third paper in this thesis, "*New dimensions in heap profiling*". Other approaches have also appeared lately.

Sansom describes a profiling scheme implemented in the Glasgow Haskell Compiler [San94]. His profiler can report both time and space used by the different parts of a program. The programmer inserts 'cost centres' to mark which parts that are of interest. (The compiler can also insert cost centres in all top level functions.) The difference between this approach and Runciman and Wakeling is mostly that resources used in "uninteresting" functions, e.g., prelude functions such that `map`, can be inherited by the user defined function that used them. The possibility to get the amount of time spent in the different functions is also new compared to Runciman and Wakeling. It is also possible to observe the amount of memory allocated, not only memory residence.

Clack, Clayman and Parrott advocate *lexical profiling* [CCP95], which they claim reflects the programmer's view of a program, as opposed to the evaluator's or implementor's views supported by other profilers. Costs are assigned to lexical units *as if* programs were executed using a 'call by value' rule, but the values reported faithfully reflect savings due to the actual use of 'call by need'. So the

scheme shares with the Glasgow profiler the ideal of attributing costs to lexical units aggregated over all their subexpressions.

Space efficient programs

One important property with functional languages is that functions are first class citizens. This fact can be used to develop higher order functions that create other functions. A nice example of this is parsing combinators [Bur75, Wad85, Hut92, PW94]. They make it easy to develop parsers, as the program structure corresponds very closely to the grammar itself, but the memory needed when using these parsers can be surprisingly large. However, it is possible to reduce this memory requirement, without reducing the usefulness of the combinators, by rewriting the implementation of them. Such a rewritten set of parsing combinators is described in the fourth paper of this thesis, "*Efficient parsing combinators*".

But sometimes the space problem is not the programmers fault: it is the compiler that introduces it. Rewriting the program will not remove the space leak, since it is the compiler that must be fixed. One example is the code produced by the Chalmers Haskell compiler if the value of a variable is returned in a function. The compiler produces code that evaluates the variable and then copies parts of the value, this avoids creating indirection nodes. (Not using indirection nodes means that dereferencing can be done faster.) Sharing is however lost which can be a problem for some programs [RW93b]. It is not possible to remove this problem by rewriting the program; the compiler must be changed. The last paper in this thesis is a description of a Haskell compiler where most decisions are resolved with the space efficient method as the winner. This compiler does not only try to produce space efficient code, it is also written to be able to re-compile itself in a very limited memory by todays standard.

The only other Haskell system that does not need large amount of memory is HUGS, a Haskell version of Gofer [Jon91, Jon94]. HUGS can however not recompile itself, as it is written in C.

Main contributions

A concurrent garbage collector for the $\langle \nu, G \rangle$-machine: One way to reduce the (observed) time for garbage collection is to do it concurrently with evaluation. This paper describes such a garbage collector, based on the Appel-Ellis-Li collector, for the $\langle \nu, G \rangle$-machine. The garbage collector temporarily stops evaluation when a garbage collection is necessary. Computation is however resumed, before the collector has finished.

The unique point is that this collector can garbage collect processes, which is quite complicated due to sharing. It also incorporates generational collection with the Appel-Ellis-Li collector. The results of using a generational collector

was surprisingly good, despite a simple implementation that could not move all kinds of nodes into the old generation.

Generational garbage collection, for lazy functional languages, without temporary space leaks: The good results for the generational collector in "*A concurrent garbage collector for the* $\langle \nu, G \rangle$*-machine*" inspired me to implement a generational collector for the G-machine. This time a *complete* generational collector was the goal, i.e., one that can tenure all kinds of nodes. The first implementation was however a disaster, suffering from enormous temporary space leaks. Small improvements to the collector removed these space leaks. The final result was a generational collector that worked well in a lazy language.

The main contribution are, beside emphasising the problem with careless tenuring in lazy languages, that the generational collector described does not need to check updates to determine if they are in the old generation, and that the method to keep track of the age of nodes is very cheap, both in space and time. Not having to check all updates also makes it possible to chose garbage collector, two-space or generational, at run-time instead of compile time. This is useful since the available real memory is not always known at compile time.

New dimensions in heap profiling: Generational garbage collectors try to decrease garbage collection overhead by concentrating on young nodes. There do however exist nodes that are even cheaper to "reclaim": the nodes that were never allocated. Heap profiling is a tool to help the programmer pinpoint, and rewrite, those parts of programs that consume lots of memory.

This work, done in collaboration with Colin Runciman, extend the previous implementation of heap profiling, from static to dynamic information. The old static profiles, which could answer "Who produced what?" are very useful, but sometimes can not find the exact problem. The question "Why are these nodes kept in memory?" is often what the programmer want to ask. Retainer profiles answers exactly this question, and have proved themselves extremely useful to reduce memory demands in lazy functional programs. Lifetime profiling is also described, and implemented, in this paper. This profile makes it possible to distinguish between nodes kept due to dragging, i.e., retained but not used, and those with a constant turn-over.

Due to this being a thesis a clarification about who did what is necessary. I suggest that the rest of this paragraph is read after reading the paper.

Colin Runciman had the original idea about retainer and lifetime profiling. The translation of these ideas into implementable algorithms was joint work, although I did the final implementation. I invented the inplace linear algorithm to derive lifetime information after Colin had described a linear algorithm that needed extra space. The sections describing the principles behind lifetime and retainer profiling were written jointly. Colin wrote the introduction, the section about clausify and most of the background and related work. I wrote about the

implementation in `nhc`, the 'nofib' Benchmark Suite and self-application to `nhc`; then Colin improved my English!

Efficient parsing combinators: This paper describes a set of space efficient and fast parsing combinators that can report the positions of errors. The combinators described have full backtracking possibilities, which makes it necessary to retain the input, but the programmer can remove the possibility to backtrack, by inserting a `cut` combinator in the parser, where backtracking cannot give a better alternative than the current one. Different variants of cut combinators exist in other sets of parsing combinators. Some of these, e.g., a combination of the `nofail` and `cut` combinators defined in [Wad85], can sometimes achieve the same improvement in speed and memory usage as the `cut` described in this paper. The unique point of my cut combinator is that it achieves the space saving and still allows reporting of errors.

`nhc` – a space-efficient Haskell compiler: This compiler tries to produce space efficient code, instead of the usual fast code. It is written to use as little memory as possible itself, and can in fact recompile itself in 3 MByte of memory, a tiny amount compared to the 12-16 Mbyte needed for other self-compiling Haskell compilers. This paper is best seen as a collection of methods in how to reduce memory consumption in compilers, and warnings about possible problems in the generated code. (Some methods are also applicable in other lazy functional programs.) The heap profiles in "*New dimensions in heap profiling*" were implemented in this compiler.

References

[App89] Andrew W. Appel. Simple generational garbage collector and fast allocation. *Software-Practice and Experience*, 19(2):171–183, 1989.

[Bur75] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley Publishing Company, Reading, Mass., 1975.

[CCP95] C Clack, S Clayman, and D Parrot. Lexical Profiling:Theory and Practice. *To appear in Journal of Functional Programming*, 5(3), 1995.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, 1981.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, 1992.

[HV88] P. H. Hartel and A. H. Veen. Statistics on Graph Reduction of SASL Programs. *Software — Practice and Experience*, 18:239–253, 1988.

[Inc94] Pure Software Inc. Purify. Manual included in distribution, Copyright 1992-1994.

[Jon91] Mark P. Jones. Release notes for gofer 2.21. Technical report, Department of Computer Science, Yale University, November 1991. Included as part of the standard Gofer distribution.

[Jon94] Mark P. Jones. The implementation of gofer functional programming system. Technical report, Department of Computer Science, Yale University, 1994.

[LH83] Henry Liberman and Carl Hewitt. A real-time garbage collector based on the lifetime of objects. *Communications of the ACM*, 23(6):412–429, 1983.

[McC60] John McCarty. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[PW94] Andrew Partridge and David Wright. Parser combinators need four values to report errors. Technical report, Department of Computing Science, University of Tasmania, 1994.

[RW93a] Colin Runciman and David Wakeling. Heap profiling of a lazy functional compiler. In John Launchbury and Patrick Sansom, editors, *Proc. 1992 Glasgow Workshop on Functional Programming*, pages 203–214. Springer–Verlag, 1993.

[RW93b] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[San94] Patrick M. Sansom. *Execution profiling for non-strict functional languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1994.

[Sew92] Julian Seward. Generational Garbage Collection for Lazy Graph Reduction. In *International Workshop on Memory Mangement*, volume 637 of *Lecture Notes in Computer Science*, pages 200–217. Springer-Verlag, 1992.

[Sin92] Duncan C. Sinclair. Lazy Wafe Graphical Interfaces for Functional Languages. Draft, Department of Computing Science, University of Glasgow, 1992.

[SJ93] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 106–116. ACM Press, June 1993.

[Ung84] David Ungar. Generational scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM Symposium on Practical Software Development Environments*, volume 32, pages 157–167, April 1984.

[Wad85] P. Wadler. How to Replace Failure by a List of Successes. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, Nancy, France, 1985.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Mangement*, Lecture Notes in Computer Science, pages 1 – 42. Springer-Verlag, September 1992.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *SIGPLAN Conference on Object Oriented Programming Systems, Languages and Implementations*, pages 23 – 35. ACM, October 1989.