Fixing some space leaks with a garbage collector*†

Philip Wadler
Programming Research Group
Oxford University
11 Keble Road, Oxford, OX1 3QD, U.K.

Abstract

Some functional programs may use more space than would be expected. A modification to the garbage collector is suggested which solves this problem in some cases. Related work is discussed.
Key words: space leak, garbage collection, functional languages, lazy evaluation.

A "space leak" is a feature of a program that causes it to use more space than one would expect. Several researchers have pointed out that space leaks are a common problem in functional languages with lazy evaluation [1, 2, 3]. In particular, space leaks may arise whenever a function uses a tuple to return more than one result. These leaks can cause a program that one would expect to run in constant space to require space proportional to the size of its input. Most disturbingly, John Hughes has pointed out that unless a parallel evaluator is used, some programs may be inherently "leaky" [1].

This paper shows that a simple modification to the garbage collector can avoid one class of space leaks. In particular, it offers an alternative to parallelism to solve the problem cited by Hughes. Similar solutions have been suggested independently by David Turner [4] and Lennart Augustsson [5].

The question of how much space one "expects" a program to use is difficult to formalize, and consequently "space leak" is not a formal term. In practice, the term has been associated with at least three different classes of

*Appeared in *Software Practice and Experience*, 17(9):595–608, September 1987.
†This work was performed under a research fellowship funded by ICL.

problems. The first class arises in connection with fully lazy supercombinators, and may be solved by not using full laziness. The second class arises in connection with accumulation functions, such as a function to sum a list of numbers, and may be solved by using strictness annotations. Both classes of leak are described in Chapter 23 of reference [3]. The third class arises in connection with functions that use tuples to return multiple results, and is the subject of this paper.

Some good quality implementations of lazy functional languages have appeared, and significant functional programs have been written using these systems. For example, the LML compiler is written in LML [6, 7], and parts of a functional text editor and operating system have been implemented on the SKIM machine [2]. As functional programming becomes more a practice as well as a theory, considerations of efficiency like those treated in this paper will become more important.

This paper is organized as follows. Section 1 presents an example of an inherently leaky program. Section 2 describes the proposed solution. Section 3 examines space leaks due to where clauses, and shows that the proposed solution fixes such leaks. Section 4 discusses related work. Section 5 presents a summary and conclusions. The Appendix outlines the code of a suitably modified garbage collector.

1 A leaky program

The following example is based on work by Hughes [1, pages 82–86]. Let *break* be a function which 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. For example,

$$\mathit{break}\ \text{“first↵next↵last”} \quad = \quad (\text{“first”}, \text{“next↵last”})$$

where the newline character is written '↵'. (The examples in this paper are written in a notation similar to Miranda[1] [8].) One way to define *break* is as follows:

$$\begin{array}{lcll} \mathit{break}\ (x : \mathit{xs}) & = & ([\,], \mathit{xs}), & \textbf{if}\ \ x = \text{‘↵’} \\ & = & (x : (\mathit{fst}\ b), (\mathit{snd}\ b)), & \textbf{otherwise} \\ & & \textbf{where}\ b \ = \ \mathit{break}\ \mathit{xs} & \end{array}$$

[1]Miranda is a trademark of Research Software Limited.

This (or something close to it) is the most natural definition of *break*, given that each element in the input is to be examined at most once.

Now consider the function *surprise*, which inserts a line saying "surprise" after the first line. For example,

$$\mathit{surprise}\ \text{“first↴next↴last”} \quad = \quad \text{“first↴surprise↴next↴last”}$$

It can be defined by:

$$\begin{array}{lcl} \mathit{surprise}\ \mathit{xs} & = & (\mathit{fst}\ b) \mathbin{+\!\!+} \text{“↴surprise↴”} \mathbin{+\!\!+} (\mathit{snd}\ b) \\ & & \quad \textbf{where}\ b \ = \ \mathit{break}\ \mathit{xs} \end{array}$$

Say that *surprise* was applied to a lazy list representing input from a file and the output list was printed to another file. One would expect this program to run in constant space, regardless of the size of the input file. But in fact this program requires space proportional to the length of the first line of its input.

To see why *surprise* will not evaluate in constant space, consider the following trace of its evaluation:

$$\begin{array}{lllcl}
\multicolumn{5}{l}{\mathit{surprise}\ \text{“first↴next↴last”}} \\
= & \multicolumn{4}{l}{(\mathit{fst}\ b_1) \mathbin{+\!\!+} \text{“↴surprise↴”} \mathbin{+\!\!+} (\mathit{snd}\ b_1)} \\
& \textbf{where} & b_1 & = & \mathit{break}\ \text{“first↴next↴last”} \\
= & \multicolumn{4}{l}{\text{‘f’} : ((\mathit{fst}\ b_2) \mathbin{+\!\!+} \text{“↴surprise↴”} \mathbin{+\!\!+} (\mathit{snd}\ b_1))} \\
& \textbf{where} & b_1 & = & (\text{‘f’} : (\mathit{fst}\ b_2), (\mathit{snd}\ b_2)) \\
& & b_2 & = & \mathit{break}\ \text{“irst↴next↴last”} \\
= & \multicolumn{4}{l}{\text{‘f’} : \text{‘i’} : ((\mathit{fst}\ b_3) \mathbin{+\!\!+} \text{“↴surprise↴”} \mathbin{+\!\!+} (\mathit{snd}\ b_1))} \\
& \textbf{where} & b_1 & = & (\text{‘f’} : (\mathit{fst}\ b_2), (\mathit{snd}\ b_2)) \\
& & b_2 & = & (\text{‘i’} : (\mathit{fst}\ b_3), (\mathit{snd}\ b_3)) \\
& & b_3 & = & \mathit{break}\ \text{“rst↴next↴last”}
\end{array}$$

The problem here is that the term $(\mathit{snd}\ b_1)$ will not be evaluated until after the entire first line has been read, assuming a sequential evaluator is used. As more and more of the first line is read, the structure pointed at by b_1 gets larger and larger.

Note that the portion of the output

$$\text{‘f’} : \text{‘i’} : \ldots$$

will already have been written, and so should not be counted as part of the space used. (But 'f' and 'i' are part of the space used, since they are still pointed at by b_1 and b_2.) Similarly, the portion of the input

$$\text{“rst↴next↴last”}$$

will not yet have been read, and also does not count as part of the space used. The space leak is due solely to the space required for representing terms like b_1 and b_2.

Of course, this only shows that one particular definition of *break* has a space leak. Hughes goes on to offer a proof that *every* definition of *break* must have a similar leak if a sequential evaluator is used. A sequential evaluator is defined to be one that, once it begins to reduce an expression E, will only reduce E and expressions demanded by E until E has been completely reduced. The proof will not be repeated here. The interestested reader may gain insight by attempting to construct a definition without a space leak, or may consult the proof in [1].

One may wonder whether this result poses a serious problem. Does it matter that there is no way of writing *break* such that *surprise* runs in constant space? After all, buffering a single line usually will not require much space. Further, it is possible to rewrite the function *surprise* in a different way, not using *break*, so that it will run in constant space.

However, a little reflection should convince the reader that this result is indeed important, because it is symptomatic of a deeper problem. In particular, *any* way of structuring a program so that some function returns a pair of results may be subject to a space leak.

2 Plugging the leak with garbage collection

One solution to the problem described above is as follows: modify the garbage collector so that whenever it encounters a term of the form

$$(\mathit{fst}\ (x, y)) \qquad\qquad \text{or} \qquad\qquad (\mathit{snd}\ (x, y))$$

then it is replaced by x or y, respectively.

With this simple modification, the definition of break given in Section 1 will now run in constant space. For example, consider again the reduction of the term

$$\mathit{surprise}\ \text{“first↵next↵last”}$$

As seen, during evaluation this becomes

$$\begin{array}{lcl} \multicolumn{3}{l}{\text{‘f’} : \text{‘i’} : ((\mathit{fst}\ b_3) +\!\!+ \text{“↵surprise↵”} +\!\!+ (\mathit{snd}\ b_1))} \\ \textbf{where}\ b_1 & = & (\text{‘f’} : (\mathit{fst}\ b_2), (\mathit{snd}\ b_2)) \\ b_2 & = & (\text{‘i’} : (\mathit{fst}\ b_3), (\mathit{snd}\ b_3)) \\ b_3 & = & \mathit{break}\ \text{“rst↵next↵last”} \end{array}$$

Say that a garbage collection occurs at this point. The collector simplifies terms of the form *fst* (x, y) and *snd* (x, y) as above, so after the collection the expression would appear as follows:

$$\begin{array}{l} \text{‘f’} : \text{‘i’} : ((\mathit{fst}\ b_3) +\!\!+ \text{“↵surprise↵”} +\!\!+ (\mathit{snd}\ b_3)) \\ \textbf{where}\ \ b_3 = \mathit{break}\ \text{“rst↵next↵last”} \end{array}$$

All references to the extra expressions causing the space leak, b_1 and b_2, have been removed by the garbage collector. It should be clear that, using this method, any application of *surprise* can be reduced using a constant amount of memory regardless of the length of the input list.

A key feature of this scheme is that nodes of the form (*fst* b) or (*snd* b) are only changed by the garbage collector if b has already been reduced to a pair by the evaluator. The evaluator is never called by the collector itself. It would be dangerous to call the evaluator from the collector, because the expression to be evaluated might use a large amount of resource or enter an infinite loop.

The modifications required to the garbage collector are relatively simple and the modified collector can be designed to run efficiently. It is often suggested that garbage collection should elide indirection nodes [9], and the modification suggested here is not much more difficult.

The implementation can be made particularly efficient if terms of the form (x, y), (*fst* b), and (*snd* b) are represented in a special way, e.g., by a node with a tag indicating that it is a pair or an application of *fst* or *snd*. Many implementations already arrange to represent pairs specially, and it would not be difficult to do the same with applications of *fst* or *snd*. As usual, the garbage collector should be designed to branch on the tag of the node. The only overhead occurs when a node tagged as an application of *fst* or *snd* is examined. In this case, the collector must examine the argument to see if it is tagged as a pair.

The code for a suitably modified collector is sketched in the appendix.

3 Where clauses

Many functional languages, such as LML and Miranda, allow "where" clauses in which the left hand side is a pattern. In these languages, an expression of the form

$$\begin{array}{lcl} E\ x\ y & & \\ \textbf{where}\ \ (x, y) & = & E' \end{array} \qquad (*)$$

is treated as equivalent to

$$\begin{array}{l} E\ (\mathit{fst}\ b)\ (\mathit{snd}\ b) \\ \textbf{where}\ b \ = \ E' \end{array}$$

where b is a new variable. Here E is an expression that may contain x and y and E' is an expression that returns a pair. For example, the definition

$$\begin{array}{lcl} \mathit{surprise}\ \mathit{xs} & = & \mathit{ys} \mathbin{+\!\!+} \text{“↵surprise↵”} \mathbin{+\!\!+} \mathit{zs} \\ & & \textbf{where}\ (\mathit{ys}, \mathit{zs}) \ = \ \mathit{break}\ \mathit{xs} \end{array}$$

is equivalent to the previous definition of *surprise*.

Without the modification to the garbage collector suggested in the previous section, this form of where clause will often cause a space leak. Say that in $(*)$ above x is fully evaluated before the value of y is required. Until y is required, it will be represented by a term of the form $(\mathit{snd}\ b)$, where the first element of b is x. So the space required by x will not be released until y is examined. This is the space leak, and it will be a serious problem whenever the space required by x is large.

A typical example of this sort of problem appears in [2, page 8.11], where Stoye notes, with understatement, that it “may have some worrying implications”. We can worry less now, since it is not hard to see that this sort of space leak is eliminated by the suggested modification to the garbage collector.

What about patterns other than tuples on the left hand side of a where clause? Consider, say, a list of length two. Analogous to $(*)$ we have:

$$\begin{array}{l} E\ x\ y \\ \textbf{where}\ [x, y] \ = \ E' \end{array}$$

where E' is now an expression that returns a list of length two. Presumably, in order to avoid a space leak in this case the garbage collector must be further modified, to reduce terms such as $(\mathit{head}\ (x : \mathit{xs}))$ and $(\mathit{tail}\ (x : \mathit{xs}))$. This is indeed one possibility. However, another solution is suggested in [3]. Peyton-Jones' solution is motivated by a completely different problem, namely, the conformality check.

When evaluating the expression above we should check that E' is indeed a list of length two; this is called a “conformality check”. It is desirable to perform this check only once, but not to perform the check unless the value of x or y is actually demanded. Peyton-Jones suggests that this be done by treating the above expression as equivalent to:

$$E\ (\mathit{fst}\ b)\ (\mathit{snd}\ b)\ \ \textbf{where}\ b \ = \ (\lambda[x, y].(x, y))\ E'$$

The conformality check will be performed only once, when the right hands side of the where clause is evaluated. (Unlike a where clause, a lambda expression may check for conformality as soon as it is applied; see [3] for further explanation.)

In general, Peyton-Jones suggests that any where clause of the form

$$\begin{array}{l} E\ v_1\ v_2 \ldots v_n \\ \textbf{where}\ p \ = \ E' \end{array}$$

should be translated to the form

$$\begin{array}{l} E\ (sel_{n1}\, b)\ (sel_{n2}\, b) \ldots (sel_{nn}\, b) \\ \textbf{where}\ b \ = \ (\lambda p.(v_1, v_2, \ldots, v_n))\ E' \end{array}$$

where p is a pattern, and v_1, v_2, $\ldots$, v_n are the variables in p. Here $(sel_{ni}\, b)$ returns the i'th component of the n-tuple b; in particular, *fst* and *snd* are abbreviations for sel_{21} and sel_{22}.

This suggests that the garbage collector should reduce all terms of the form

$$(sel_{ni}\,(x_1, x_2, \ldots, x_n))$$

to x_i. This is a straightforward generalization of the previous suggestion. Additional reduction rules, for selectors such as *head* and *tail*, are not necessary.

Perhaps it seemed to the reader that the introduction of reduction rules in the garbage collector for *fst* and *snd* was a bit *ad-hoc*. Why not add other reduction rules, say for *head* and *tail*, or for other common functions? The purpose of this section was to show that adding reduction rules to the collector could be motivated in terms of avoiding space leaks due to where clauses. From this point of view (and assuming Peyton-Jones' translation of where clauses) it seems that a choice to add reduction rules just for the selector functions sel_{ni} is appropriate.

4 Relation to other work

As mentioned, Hughes has proved that *any* definition of *break* cannot run in constant space. This paper has shown that a simple modification to the garbage collector will allow *break* to run in constant space. Does this mean that Hughes' proof is incorrect? Not at all. Hughes showed that *break* could not be evaluated in constant space by a sequential evaluator: one that only reduces one expression (and its sub-expressions) until it has been reduced to normal form. The modification described here is not sequential, in that if garbage collection occurs during reduction of some expression then other expressions (that are not its sub-expressions) may be reduced by the collector.

The method described here is less expensive to implement than the parallelism advocated by Hughes. It also has other advantages. Hughes suggests that the source text be modified to indicate where parallel evaluation and synchronisation should occur, whereas the method suggested here requires no change to the source text. Further, Hughes' method is unsafe: injudicious use of parallelism can increase the time or space complexity of a program and, worse, injudicious use of synchronisation can cause deadlock (making a program undefined). In contrast, the method described here is completely safe: it cannot lead to deadlock, it leaves the time complexity of the program unchanged, and it leaves the space complexity unchanged or improved.

On the other hand, Hughes' method also supports program structures that are not supported by the method described here. For example, an alternative method of defining *break* is as follows:

$$\mathit{break}\ \mathit{xs} \quad = \quad (\mathit{before}\ \text{`↵'}\ \mathit{xs}, \mathit{after}\ \text{`↵'}\ \mathit{xs})$$

Here (before '↵' xs) returns the portion of the list xs before the first '↵' and (after '↵' xs) returns the portion after it. One can argue that this definition of *break* is clearer than the preceding one, because it divides the problem into two parts, *before* and *after*, which are smaller and have general utility.

Using Hughes' proposal, this definition of *break* can be made to work in constant space by adding parallelism and synchronization operators:

$$\begin{array}{lcl} \mathit{break}\ \mathit{xs} & = & (\mathit{PAR}\ \mathit{before}\ \text{`↵'}\ \mathit{ys}, \mathit{PAR}\ \mathit{after}\ \text{`↵'}\ \mathit{zs}) \\ & & \quad \textbf{where}\ (\mathit{ys}, \mathit{zs}) \ = \ \mathit{SYNCHLIST}\ \mathit{xs} \end{array}$$

This is uglier than the preceding definition, but still has essentially the same structure. The *SYNCHLIST* operator causes ys and zs to be bound to copies of the input list xs. They are synchronized so that an element of the input list xs is not read until (before '↵' ys) requires the next element of ys and (after '↵' zs) requires the next element of zs. The *PAR* operators cause these two expressions to be evaluated in parallel. If *SYNCHLIST* were used here without *PAR*, then deadlock might result. The reader is referred to [1] for more details.

The modification to the garbage collector described here does not allow the alternative definition of *break* to run in constant space. So although the solution suggested here is simpler than Hughes', it does not address as wide a range of program structures. However, an alternative solution for this particular program is mentioned below.

The results of this paper are also relevant to a different piece of work, the listless transformer [10, 11, 12]. Roughly speaking, the listless transformer will take any program which can be evaluated in constant space and transform it to an equivalent listless program. The advantage of doing this is that a listless program examines each input list only once and builds no intermediate lists.

The listless transformer benefits from the modification described here in two ways. First, the modification allows more programs to run in constant space, so the listless transformer will apply to more programs. To take into account the change to the garbage collector, one simply modifies the symbolic evaluator of the listless transformer to also perform any reductions that the garbage collector can perform.

For example, the function *break*, as defined above in terms of *before* and *after*, will count as listless once the modification to the garbage collector is included. (In the original work, functions such as *break* were made listless by including annotations to indicate that extra reductions should be performed. This was called "languid evaluation"; see Chapter 7 of reference [11]. The modification described here is a much simpler method of achieving this effect.)

The second advantage is as follows. There is an easy way of translating a listless program into an equivalent functional program; however, this translation involves "where" clauses with tuples on the left-hand side. So for the reasons described in Section 3 the translated program may not run in constant space, unless the modification described here is used.

For example, if the listless transformer is run on the definition of *break* in terms of *before* and *after*, and the resulting listless program is translated back into a functional program, then one gets the efficient definition of *break* given in Section 1. Unlike Hughes' method, the program need not be changed to indicate parallelism and synchronisation, and the run-time program is more efficient. On the other hand, the listless transformer, like the garbage collection method, does not have as wide a scope of application as Hughes' method.

In [12], it is shown how to integrate a listless program directly into a graph reducer. An alternative, made possible by the modification to the garbage collector, is to simply translate listless programs into equivalent functional programs. Which of these is better remains to be seen, since listless programs may still have fewer overheads than the equivalent functional programs. In particular, listless programs don't need to build tuples, while the equivalent functional programs do.

5 Conclusions

To summarize: Hughes has pointed out that some programs which one expects to run in constant space cannot do so under a sequential evaluator; other researchers have reported similar problems. This paper proposes a modification to the garbage collector as a solution to this problem. The modification eliminates a class of space leaks due to where clauses.

Hughes has suggested parallelism as a solution to the same problem. The method suggested here is considerably simpler than Hughes', but does not apply to as wide a range of problems. The method suggested here also improves the performance of the listless transformer. Coincidentally, the listless transformer also provides a better solution for some of the problems addressed by Hughes' method, but again does not apply to as wide a range.

The modification suggested here would not be very helpful if it improved efficiency in a haphazard way that was difficult to predict. Fortunately, it does not suffer from this drawback. The modification to the implementation is relatively simple, namely, the garbage collector is changed to perform a certain class of reductions when possible. The modification to the model of computation (graph reduction) is correspondingly simple: one merely does not count space occupied by redexes in the class reduced by the garbage collector.

A theory that allows one to reason about the space and time efficiency of functional programs is long overdue. One reason that such a theory has been late in appearing is that garbage-collection and lazy evaluation can make it difficult to predict the run-time behaviour of functional programs. In particular, currently only *ad hoc* methods are available to determine whether a program suffers from space leaks. The method suggested here does not make this problem any harder, but it does not make it much easier, either.

However, the method does have two important advantages. First, if it is used then in practice many fewer programs will have such leaks. Second, if one's program does suffer from a space leak, then this method may make it easier to rewrite the program so that the leak does not appear. As Hughes' work has shown, without this method (or some similar method) it may be impossible to eliminate the leak without completely changing the structure of the program.

In short, the method does not make it easier to discover when one's program is leaking. But it does have the advantage that in practice fewer programs will spring leaks. Further, the method makes it possible to fix the leak by tightening a joint, so to speak, whereas without it one might be forced to put in a whole new set of plumbing.

Acknowledgement. I am grateful for comments from David Lester and the referees.

Appendix: The Garbage Collector

This appendix gives an outline of the structure of the modified garbage collector. The code sketches are written in Modula-2 [13]. The appendix considers the mark phase of a simple mark-scan garbage collector with a recursive mark procedure. Other kinds of collector are discussed briefly at the end.

The modified collector reduces terms of the form *fst* (x, y) and *snd* (x, y) to x and y, respectively. It also elides indirection nodes.

As mentioned in the text, we assume that there exist special nodes to represent indirect references, the pair constructor, and applications of the functions *fst* and *snd*. These are declared in Figure 1. The ellispses in the declaration represent all of the other node types, such as nodes to represent function applications and atomic values.

The central idea is to introduce a routine `Simplify`, shown in Figure 2. This routine is a simplified version of the evaluator that only performs the reductions associated indirection nodes and applications of *fst* and *snd* to pairs. If it is possible to perform a reduction, then `Simplify(n)` sets `n` to point at the resulting node; otherwise, `n` is left unchanged. In addition, if a node representing *fst* (x, y) is reduced, then it is overwritten with an indirection node pointing at x. This guarantees that each reduction will be performed at most once. Terms of the from *snd* (x, y) are handled similarly[2].

A subtle issue is the avoidance of infinite loops. These might arise from a cycle involving only `INDIR`, `FST`, `SND`, and `PAIR` nodes. Such a cycle might arise, for instance, if an indirection node pointed at itself, or from a where

[2] An earlier version of this paper contained a bug. A node representing *fst* (x, y) was overwritten with a *copy* of the node representing x, rather than an indirection node pointing at x. This could cause the evaluation of x to be duplicated. Such overwriting bugs are surprisingly common in evaluators.

```
TYPE  Tag  =  (INDIR, PAIR, FST, SND, ...)

TYPE  Node  =
  RECORD
    mark : BOOLEAN;
    CASE tag: Tag OF
        INDIR:  ref: ^Node;
      | PAIR:   fst, snd: ^Node;
      | FST:    arg: ^Node;
      | SND:    arg: ^Node;
      ...
    END;
  END;
```

Figure 1: Declarations for garbage collector

clause of the form

$$\ldots \textbf{ where } x \; = \; \mathit{fst}\ (x, y)$$

To avoid such loops, `Simplify` is designed to do nothing if the mark bit is set, to set the mark bit before it makes any recursive call, and to reset the mark before exit. This guarantees that any call of `Simplify` will terminate, and leaves all mark bits unchanged after the call.

Although this is adequate, less marking need be done if it can be guaranteed that certain kinds of cycles cannot arise. There are four kinds of cycles of interest. First is a cycle consisting only of indirection nodes. Most systems are designed so that such cycles arise only if the evaluator is already in an infinite loop. Hence, it is safe to ignore such cycles. Second is a cycle consisting only of `INDIR`, `FST`, and `SND` nodes, but no `PAIR` nodes. Such a cycle might arise from a where clause of the form

$$\ldots \textbf{ where } x \; = \; \mathit{fst}\ x$$

In the Hindley/Milner polymorphic type system used in LML and Miranda, this declaration is ill-typed, as are all expressions that could give rise to such a cycle. Hence, under such a type system, it is again safe to ignore such cycles. Third is a cycle consisting only of `INDIR` and `PAIR` nodes. Again, in the Hindley/Milner type system such cycles cannot arise. Fourth is a cycle

```
PROCEDURE Simplify (VAR n:^Node);
  VAR
    k:^Node;
  BEGIN
    IF NOT n^.mark THEN
      n^.mark := TRUE;
      CASE n^.tag OF
          INDIR:
            Simplify (n^.ref);
            n := n^.ref;
        | FST:
            Simplify (n^.arg);
            k := n^.arg;
            IF k^.tag = PAIR THEN
              Simplify (k^.fst);
              n^.tag := INDIR;
              n^.ref := k^.fst;
              n := n^.ref;
            END;
        | SND:
            (* similar to FST *)
            ...
        ELSE
            (* do nothing *)
      END;
      n^.mark := FALSE;
    END;
  END Simplify;
```

Figure 2: Routine `Simplify`

```
PROCEDURE Simplify2 (VAR n:^Node);
  VAR
    k:^Node;
  BEGIN
    IF NOT n^.mark THEN
      CASE n^.tag OF
          INDIR:
            Simplify2 (n^.ref);
            n := n^.ref;
        | FST:
            Simplify2 (n^.arg);
            k := n^.arg;
            IF k^.tag = PAIR THEN
              n^.mark := TRUE;
              Simplify2 (k^.fst);
              n^.mark := FALSE;
              IF n = k^.fst THEN k^.fst := Loop END;
              n^.tag := INDIR;
              n^.ref := k^.fst;
              n := n^.ref;
            END;
        | SND:
            (* similar to FST *)
            ...
        ELSE
            (* do nothing *)
      END;
    END;
  END Simplify2;
```

Figure 3: Improved version of `Simplify`

which involves at least one `FST` or `SND` node and one `PAIR` node. We saw above an example of a where clause that gives rise to such a cycle.

Figure 3 shows a version of `Simplify` modified under the assumption that cycles of the first three kinds do not arise, while cycles of the fourth kind may. Thus, it is no longer necessary to set the mark bit while simplifying an indirection node or the argument of a `FST` or `SND` node, although it is necessary to set the mark bit of a `FST` or `SND` node before simplifying the corresponding field of a `PAIR` node.

It is also necessary to ensure that no new cycles of indirection nodes are created by the simplifier. Since `Simplify2` elides chains of indirection nodes, such cycles could only consist of a single indirection node pointing at itself. In Figure 3, the test `n = k^.arg` checks whether such a cycle would be formed. If so, it replaces the relevant pointers with `Loop`, which is assumed to point to a special, pre-allocated node that denotes an infinite loop.

Thus, if cycles of the second and third kind are ill-typed, and the initial structure does not contain cycles of indirection nodes, then `Simplify2` is guaranteed not to terminate and not to introduce such cycles. A key step in proving this is to verify that after `Simplify2` terminates its argument never points to an indirection node, so all chains of indirection nodes are elided, and no cycles of indirection nodes are formed.

We now turn to the design of the mark routine itself. Figure 4 shows a straightforward mark routine, `Mark(n)`. This routine first calls `Simplify2(n)`, and then ensures that `n` and all of its descendants are marked. The ellipses refer to the cases for the other node types, which are similar to the cases shown.

This implementation is less efficient than it might be, since both `Simplify2` and `Mark` use a case statement to examine the same node. Performance can be improved by merging the case statements in `Simplify2` and `Mark`; the result of doing this is shown in Figure 5. This routine incurs the overhead of calling `Simplify2` only when it is passed a `FST` or `SND` node.

This concludes the outline of the mark phase of a mark-scan garbage collector; the scan phase is not affected.

Essentially the same program structure as above will suffice for a "two-space" copying collector. Instead of simply setting the mark bit of a node, the mark routine must be modified to allocate a new node (in the "to" space), copy into it the old node (in the "from" space), overwrite the old node with an indirection referring to the new node, and return the address of the new node.

```
PROCEDURE Mark (VAR n:^Node);
  BEGIN
    Simplify2 (n);
    IF NOT n^.mark THEN
      n^.mark := TRUE;
      CASE n^.tag OF
          INDIR:
            (* can't happen *)
        | FST:
            Mark (n^.arg);
        | SND:
            Mark (n^.arg);
        | PAIR:
            Mark (n^.fst);
            Mark (n^.snd);
        | ...
      END;
    END;
  END Mark;
```

Figure 4: Routine `Mark`

```
PROCEDURE Mark2 (VAR n:^Node);
  VAR
    k : ^Node;
  BEGIN
    IF NOT n^.mark THEN
      CASE n^.tag OF
          INDIR:
            Mark2 (n^.ref);
            n := n^.ref;
        | FST:
            Simplify2 (n^.arg);
            k := n^.arg;
            IF k^.tag = PAIR THEN
              n^.mark := TRUE;
              Mark2 (k^.fst);
              n^.mark := FALSE;
              IF n = k^.fst THEN k^.fst := Loop END;
              n^.tag := INDIR;
              n^.ref := k^.fst;
              n := n^.ref;
            ELSE
              n^.mark := TRUE;
              Mark2 (n^.arg);
            END;
        | second:
            (* similar to FST *)
            ...
        | pair:
            n^.mark := TRUE;
            Mark2 (n^.fst);
            Mark2 (n^.snd);
        | ...
      END;
    END;
  END Mark2;
```

Figure 5: Improved version of `Mark`

It is not clear whether the program structure outlined above will apply to other more ingenious forms of garbage collector, such as those which implement a stack by reversing pointers. But it is hoped that, with sufficient additional ingenuity, these too could be modified in the desired way.

References

[1] John Hughes. *The design and implementation of programming languages*. PhD thesis, Oxford University, July 1983. Programming Research Group, technical monograph PRG-40.

[2] William Stoye. *The implementation of functional languages using custom hardware*. PhD thesis, Cambridge University, December 1985. Computing Laboratory, technical report 81.

[3] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[4] David Turner. A proposal concerning the dragging problem. October 1985. Burroughs ARC internal report.

[5] Lennart Augustsson. Private communication.

[6] Lennart Augustsson. A compiler for Lazy ML. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[7] Thomas Johnsson. Efficient compilation of lazy evaluation. In *ACM SIGPLAN Symposium on Compiler Construction*, June 1984. *SIGPLAN Notices* **19**(6).

[8] David Turner. Miranda: a non-strict language with polymorphic types. In *Symposium on Functional Programming Languages and Computer Architecture*, Nancy, France, Springer-Verlag, September 1985. Lecture Notes in Computer Science 201.

[9] David Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9(1), January 1979.

[10] Philip Wadler. Listlessness is better than laziness: lazy evaluation and garbage collection at compile-time. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[11] Philip Wadler. *Listlessness is better than laziness.* PhD thesis, Carnegie-Mellon University, August 1984.

[12] Philip Wadler. Listlessness is better than laziness II: composing listless functions. In *Workshop on Programs as Data Objects*, Copenhagen, Springer-Verlag, October 1985. Lecture Notes in Computer Science 217.

[13] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 1982. Third corrected edition, 1985.