Lazy Memo-functions

John Hughes
Programming Methodology Group
Department of Computer Science
Chalmers University of Technology
S-412 96 Göteborg, Sweden
uucp: ..!mcvax!enea!chalmers!john

1. Introduction

In this paper we introduce a slight variation on the old idea of a memo-function. We call our variant "lazy memo-functions" since they are suitable for use in systems with lazy evaluation. Lazy memo-functions are much more useful than the conventional kind. They can be used to define functions on cyclic structures in a natural way, and to allow modular versions of algorithms to be used when the only efficient alternative is to destroy the algorithm's structure. They can be implemented more efficiently than ordinary memo-functions, and can be used to define ordinary memo-functions if these are really required.

Memo-functions were originally invented by Michie [14]. The idea behind them is very simple: a memo-function is like an ordinary function, but it remembers all the arguments it is applied to, together with the results computed from them. If it is ever re-applied to an argument the memo-function does not recompute the result, it just re-uses the result computed earlier. "Memoisation" is an optimisation which replaces a potentially expensive computation by a simple table look-up.

The classic example of a function which can be improved by memoisation is the Fibonacci function.

```
fib n = 1, if n<2
fib n = fib(n-1) + fib(n-2), otherwise
```

Since each call of fib generates two recursive calls with smaller arguments, the cost of computing the nth Fibonacci number in this way is exponential in n. Yet if fib is memoised, so that (fib n) is computed only once for each value of n, then the nth Fibonacci number can be computed in linear time. We will indicate memoisation by prefixing the equations defining a function by the keyword memo.

```
memo fib n = 1, if n<2
memo fib n = fib(n-1) + fib(n-2), otherwise
```

Memoisation can make an enormous difference to the efficiency of an algorithm. As in this case, the difference can be so large that it is impractical to use the algorithm without it. Indeed, Turner has remarked that memoisation is a generally useful tool for developing programs: often an elegant algorithm with exponentially bad performance can be converted into an efficient one by memoisation [20]. Bird reports that program development by transformation often results in an algorithm that would be efficient if it were memoised [3]. Keller and Lindstrom argue that "mathematical elegance in problem specifications can invite [needless function recomputation]"

[12]. Yet memoisation is rarely used in practice. Instead, such algorithms are transformed further so that the need for memoisation disappears. This involves the programmer in more work, and often destroys the modular structure of the algorithm, making it hard to understand and hard to modify.

Why are memo-functions so rarely used in practice? One reason is that they are hard to implement efficiently when the argument to the memo-function is a compound data-structure. The memo-function must compare each argument against the arguments in its memo-table, and comparing data-structures for equality is expensive (a recursive procedure is required). It is difficult to apply sophisticated techniques such as hashing to find equal structures. The alternative of storing all data-structures uniquely (using a hashing cons, see section 5) so that equality can be tested by a simple pointer comparison is even less attractive because it imposes an overhead on the creation of all data-structures, whether they are ever passed as arguments to memo-functions or not. In a language with lazy evaluation this problem is aggravated: since verifying that two data-structures are equal requires that each be completely evaluated, all memoised functions are completely strict. This means they cannot be applied to circular or infinite arguments, or to arguments which (for one reason or another) cannot yet be completely evaluated. Therefore memo-functions cannot be combined with the most powerful features of lazy languages. Finally, memo-functions can interfere with garbage collection and cause otherwise harmless programs to run out of space. This is because the memo-table in which arguments and results are stored always grows. Even if an argument will never be passed to a memo-function again, it and the associated result remain in the memo-table taking up space. The implementation cannot know when it is safe to delete such entries from the memo-table.

2. **Lazy Memo-functions**

Our variant, lazy memo-functions, is intended to be used with lazy evaluation. It addresses all the problems discussed above to a greater or lesser extent. The basic idea is to weaken the requirement a memo-function must satisfy. Ordinary memo-functions are required to re-use previously computed results if applied to arguments equal to previous ones. Lazy memo-functions need only do so if applied to arguments identical to previous ones - that is, arguments stored in the same place in memory. In Lisp terms, we are using EQ to test for repeated arguments, rather than EQUAL. Two objects are tested for identity as follows:

(1) If they are stored at the same address, they are identical. Return true.
(2) If they are atomic values (such as numbers, booleans, characters) they are identical if they are equal.
(3) Otherwise they are not identical. Return false.

(We test atomic values for equality because, not being data-structures, they have no address to compare. Even if atoms are stored as data-structures it is essential that equal atoms be identical to provide a base-case for unique, defined in section 5).

This is much more efficient than a potentially recursive equality test. In addition, memo-tables can be indexed by a hash value computed from the address of the stored argument, making memo-lookup more efficient. Hash techniques are hard to use with conventional memo-functions because equal structures must have the same hash value.

Also we can see from the definition that the components of a compound object (such as a CONS cell) are not involved when the object is tested for identity. Thus in a lazy language it is possible to test partially evaluated objects for identity. The arguments to a lazy memo-function must be evaluated far enough to discover which object they refer to, but the components of that object need not be evaluated. Thus lazy memo-functions are strict only to one level. They combine well with lazy evaluation.

Finally, it is possible to delete some entries from lazy memo-tables without risking repeating computations. When an argument stored in a memo-table is reclaimed by the garbage collector it can also be deleted from the memo-table, since it can never reappear as an argument to any function. Implementing this requires some care, though, since references from memo-tables themselves must not be counted. We will return to this in section 6.

Lazy memo-functions can be implemented in terms of an identity test, hash tables, and assignment. Morris has suggested adding an identity test and hash tables directly to a functional language [15]. Unfortunately this compromises referential transparency, since equal structures cannot be substituted for equals in identity tests without changing the meaning. For example

```
(lambda x. identical x x) (cons 1 2)
```

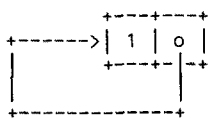
evaluates to true, but the supposedly equivalent

```
identical (cons 1 2) (cons 1 2)
```

evaluates to false, since the two calls of cons construct cells at different addresses. Lazy memo-functions provide some of the same power as Morris' primitives, and do not compromise referential transparency at all.

3. Functions on Cyclic Structures

The most striking application of lazy memo-functions is to the manipulation of cyclic structures. These are used to represent infinite objects in a compact and efficient way. For example, the infinite list of ones can be represented by the cyclic structure



The programmer creates cyclic structures using recursive declarations. For example, the structure above would be created by the declaration

```
ones = 1:ones
```

(where ":" is the "cons" operator which adds an element to the front of a list. Our notation in this paper is based on Turner's language KRC [21] with a few liberties taken here and there).

Unfortunately cyclic structures are not "first-class citizens", in the sense that any manipulation of a cyclic structure is likely to result in an infinite one. For example, one might try to define the infinite list of twos by

```
twos = map double ones
double x = 2*x
```

where map returns the list obtained by applying double to each element of ones. This definition makes twos a non-cyclic, infinite list of twos. Essentially this is because there is no way for a function to distinnguish a cyclic representation from a truly infinite one of the same data-structure, and therefore (map double) must return the same result whichever it is applied to. When it is applied to a truly infinite list, it cannot possibly predict that all the elements of the list will be one, and so it cannot possibly return a cyclic structure. Therefore it must return an infinite result even when its argument is cyclic.

This means that cyclic structures must be manipulated with great care if they are not to become infinite. It is important to keep structures cyclic, not only because cyclic structures are much more compact than infinite ones, but because finite structures take only a finite amount of work to build. Once completed they can be accessed freely. An infinite structure, on the other hand, consumes more and more computer time as more and more of it is created.

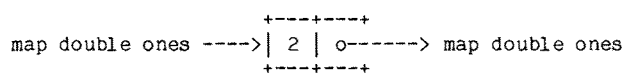
Lazy memo-functions can help keep structures cyclic. To see why, let us return to (map double ones). The definition of map is

```
map f [] = []
map f (a:x) = f a:map f x
```

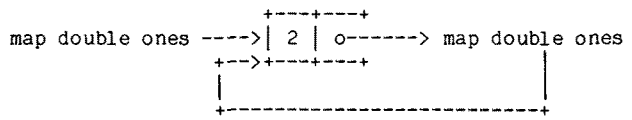
(In KRC lists are written down by enclosing their elements in square brackets, so [] represents the empty list), and so

```
map double ones = map double (1:ones)          (since ones=1:ones)
                = double 1:map double ones
                = 2:map double ones
```

So (map double ones) evaluates to the structure



The arguments of the recursive call of map are identical to the arguments of the first call. Therefore if map is memoised, the original result will be re-used, producing the structure



Since the recursive call forms part of the result of the first call, a cyclic structure is created. (Cycles of length one are recognised and treated in the same way in Daisy [22]). The new definition of map is

```
memo map f [] = []
memo map f (a:x) = f a:map f x
```

So simply by writing the keyword memo in front of the equations defining map we can convert it into a function which produces cyclic results from cyclic arguments. (We adopt the convention that memo memoises a function with respect to all the arguments appearing in the memoised equation, so this definition of map is memoised with respect to both arguments. Map and (map f) are both memo-functions. We do not consider whether any meaning can be given to definitions in which some equations are prefixed with memo and others are not).

Another interesting example is the function zip, which takes two lists and constructs a list of pairs of corresponding elements. For example, (zip [a,b,c] [1,2,3]) is [[a,1],[b,2],[c,3]]. Zip can be defined by

```
memo zip [] [] = []
memo zip (a:x) (b:y) = [a,b]:zip x y
```

Now consider the cyclic lists

```
ab = a:b:ab
abc = a:b:c:abc
```

Zipping these together gives

```
zip ab abc = zip (a:b:ab) (a:b:c:abc)
           = [a,a]:zip (b:ab) (b:c:abc)
           = [a,a]:[b,b]:zip ab (c:abc)
           = [a,a]:[b,b]:[a,c]:zip (b:ab) abc
           = [a,a]:[b,b]:[a,c]:[b,a]:zip ab (b:c:abc)
           = [a,a]:[b,b]:[a,c]:[b,a]:[a,b]:zip (b:ab) (c:abc)
           = [a,a]:[b,b]:[a,c]:[b,a]:[a,b]:[b,c]:zip ab abc
```

At this point zip is applied to arguments identical to those of the first call, and a cyclic structure is constructed. This structure could also be defined by

```
z = [a,a]:[b,b]:[a,c]:[b,a]:[a,b]:[b,c]:z
```

Six list cells are required to represent it (not counting cells used to represent pairs [a,a] etc.). This is the product of the number of cells used to represent each argument.

On the other hand, if ab is zipped together with itself, we get

```
zip ab ab = [a,a]:zip (b:ab) (b:ab)
          = [a,a]:[b,b]:zip ab ab
```

and so a cyclic structure with only two cells is created. In fact, the number of cells in zip's result is the least common multiple of the number of cells in each argument. This behaviour is not trivial to predict, but the memo-mechanism automatically makes the results as small as possible.

4. Structuring Functions on Trees

4.1 A Simple Example

Lazy memo-functions can also be used to make modular versions of algorithms efficient. As a very simple example of this, consider the problem of finding the deepest leaves in a tree. Since there may be many leaves at the same depth, the result must be a list of leaves. Assuming that nodes in the tree are either (leaf x) or (node l r), we can define

```
deepest (leaf x) = [x]
deepest (node l r) = deepest l, if depth l>depth r
                   = deepest r, if depth l<depth r
                   = deepest l ++ deepest r, otherwise
```

(where ++ appends two lists together). We must also define the function depth:

```
depth (leaf x) = 0
depth (node l r) = 1 + max (depth l) (depth r)
```

As it is written this algorithm is very inefficient, because the depth of each node is computed many times. The deepest parts of the tree have their depth computed as many times as there are levels in the tree. This makes the total cost of the algorithm quadratic in the size of the tree. All that is necessary to make it efficient is to memoise the function depth. Then the depth of each node is computed only once, and the whole algorithm requires time linear in the size of the tree.

Deepest can be computed efficiently in other ways. For example, an extra component could be added to every node to hold the depth of the subtree below it. This technique of storing additional data in trees is commonly used in imperative programs. However, it's rather unsatisfactory here. Because functions are written using pattern matching, every function on trees must "know about" the extra component, even though it is often irrelevant. Responsibility for maintaining the depth information is divided among all the functions that construct trees, instead of localised in one place (depth). There is therefore greater scope for error. Finally, trees with added depth information are a different type from ordinary trees, so functions that are common to both types must be duplicated. Memoising depth can be thought of as adding a "depth" field to those trees that need one, and in fact it has very similar space requirements. However, this is achieved in a transparent way and so the accompanying problems are avoided.

An alternative method is to combine the functions depth and deepest into one function with two results. This is the conventional "functional" solution, and it is equally efficient. However, the resulting function is large and complex compared to the original two. Memoising depth is a simpler, more modular solution.

Adding extra fields to data structures to hold derived values, and combining several functions into one with several results, are both common programming techniques. As in this simple example, memoisation can often offer a more elegant alternative.

4.2 **An Interpreter for the Lambda-calculus**

Compilers and interpreters provide many examples where lazy memo-functions can be applied. They manipulate syntax trees, and often need to attach derived information to the nodes. We will discuss several examples of this type. The first is an interpreter for the lambda-calculus. Lambda-expressions are represented by syntax trees with three different kinds of node,

```
<exp> ::= id <string> | app <exp> <exp> | lam <string> <exp>
```

The interpreter is a function called eval which takes a lambda-expression and reduces it to head normal form (HNF). An expression is in head normal form if it is not an application of a lambda-expression. There is no requirement that the parts of an expression in HNF be in HNF themselves. It follows that an expression in HNF must be either a lambda-expression (whose body may be in any form), or an application of an identifier to zero or more arguments (which may be in any form). Reduction to HNF is akin to lazy evaluation of functional programs, because the "components" of a result are not evaluated when the result is.

Eval can be defined by

```
eval (app f x) = apply (eval f) x
eval E = E, otherwise

apply (lam x e) a = eval (subst x a e)
apply f a = app f a, otherwise
```

where subst is a function which substitutes a for x wherever it occurs in the syntax tree e. This interpreter quite correctly does not try to evaluate arguments to functions before substituting them into the function body. If it did so, it would risk going into an infinite loop, since the argument might be an expression which has no HNF and which causes eval to loop infinitely looking for it. If E is such an expression then (app (id"f") E), for example, has a HNF - itself - which eval finds. If eval tried to evaluate the argument E it would fail to find this HNF. There is a penalty for this: arguments which are used several times in a function are re-evaluated each time they are used. If eval is memoised this causes no problem. Without memoisation it makes the evaluator far too inefficient to be used - some expressions take exponentially longer to be evaluated.

In this example the memoisation mechanism mimics the ordinary lazy evaluation mechanism. Just as the latter records the value of an expression the first time it is

computed so that subsequent evaluations do no work, so the memoised eval records the value of a node the first time it is computed, and never redoes the work. One can even add lazy memo-functions to the interpreted language by adding a new kind of lambda-expression, (memo <string> <exp>), and a new equation to the definition of apply:

```
apply (memo x e) a = apmemo (memo x e) (eval a)
memo apmemo (memo x e) a = eval (subst x a e)
```

Since apmemo is itself memoised, it never applies the same memo-function to the same argument more than once, it just reuses the old result. The effect of this is that the interpreted memo-function is itself memoised.

4.3 **An Expert System Evaluator**

Our second example of a "compiler/interpreter" is a logical expression evaluator for an expert system. This is a simplified version of one of the evaluators discussed in [4] which in turn was based on the evaluator in MYCIN [17]. The logical expressions are represented by syntax trees containing and, or and not nodes.

```
<exp> ::= and <exp> <exp> | or <exp> <exp> | not <exp> | ask <string>
```

At the leaves are "ask" nodes, which are evaluated by asking the user the question in the node and using the reply (true or false) as the value. The result of a logical expression is used to decide whether or not some hypothesis is true. For example, one might use the expression

```
(or (not (ask "Will it start?")) (ask "Does it sound nasty?"))
```

to decide whether or not there is anything wrong with an engine. Given an association list of questions and answers (qas), such a logical expression can be evaluated by the function eval defined by

```
eval qas (ask q) = assoc qas q
eval qas (and a b) = eval qas a & eval qas b
eval qas (or a b) = eval qas a | eval qas b
eval qas (not a) = ~eval qas a
```

The questions which the user should be asked can be found using the function questions

```
questions (ask q) = [q]
questions (and a b) = union (questions a) (questions b)
questions (or a b) = union (questions a) (questions b)
questions (not a) = questions a
```

where union is used to ensure that the same question is not asked more than once. Now if the user's reply (a list of booleans) is zipped together with the list of questions we get the association list of questions and answers. Therefore we can define a function expert, which takes the user's input and a logical expression as arguments and returns a list of questions and a boolean result as follows:

```
expert e input = [qs, ans]
    where qs = questions e
          qas = zip qs input
          ans = eval qas e
```

This "expert system" leaves a lot to be desired, because it always asks the user all the questions in the tree. It would be far better to omit questions whose answer is irrelevant. For example, when evaluating the expression

```
(or (ask "Do you drink?") (ask "Do you smoke"))
```

it is unnecessary to ask the user if he smokes if he has already admitted to drinking, because the expression will evaluate to true in any case. Indeed, since the KRC '|' operator does not evaluate its right argument if the left one is true, eval will not even be applied to the second question. To take this into account, we define

```
questions qas (and a b) =
        = union (questions qas a)
                (includeif (eval qas a=true) (questions qas b))
questions qas (or a b) =
        = union (questions qas a)
                (includeif (eval qas a=false) (questions qas b))
includeif true set = set
includeif false set = []
```

(qas has to be supplied as an additional argument to questions since it now calls the evaluator). This modification makes the next question asked depend on the answers to the previous ones - the questions in the second branch of an or, for example, are asked only if the first branch evaluated to false. While this certainly works without memoisation it works very badly, since some expressions may be evaluated very many times in order to decide which questions to ask. To make it acceptably efficient eval must be memoised. This has the added advantage that if parts of the tree are shared between different branches then they are only evaluated once.

4.4 **Incremental Compilation**

An incremental compiler stores a representation of the user's program, together with its compiled code. It allows the user to edit his program, and recompiles as little as possible when changes are made. Keeping track of exactly which information need be recomputed after a change can be difficult. Memo-functions can be used to do this automatically. If an ordinary compiler is modified by adding a structure editor to allow the syntax tree to be edited, and memoising the main code generation functions, then it will work as an incremental compiler. After each change the code generator can be applied to the entire syntax tree, and new code will be generated only for those parts that have changed.

4.5 **A Compiler to Super-combinators**

Our final example is a maximal free expression abstractor due to Bird [2]. Such an abstractor translates lambda-expressions into "super-combinators" [7]. It is based on the principle that, if a sub-expression E of a lambda-expression does not contain the bound variable (i.e. it is a free expression) then it can be "abstracted out".

```
(lam x ...E...) => (app (lam a (lam x ...a...)) E)
```

where a is a new identifier. This transformation improves the program because the expression E is only evaluated once, rather than every time the function containing it is called. If all the maximal free expressions are abstracted out from lambda-expressions then the resulting program is "fully lazy" - it exhibits self-optimising properties first observed by Turner [19]. The lambda-expressions which remain have no free variables, and can therefore be implemented very efficiently [10]. Such lambda-expressions are called super-combinators.

The lambda-expressions have the same structure as those in our interpreter for the lambda-calculus, except that the strings in identifier nodes are replaced by integers, being the nesting depth at which they are bound. Thus the most global identifier is replaced by 1, the next most global by 2 and so on. The translator can be defined by

```
trans (id n) = id n
trans (app a b) = app (trans a) (trans b)
trans (lam n b) = abstract (mfes n b') n b'
    where b' = trans b
```

Assuming that (mfes n b') returns a list of the maximal subexpressions of b' which do not contain the variable n, abstract replaces them by new parameters of the lambda-expression.

```
abstract m n b = mkapp m (mkcom (m++[id n]) b)
mkapp [] f = f
mkapp (a:m) f = mkapp m (app f a)
mkcom m b = com (length m) (subst m [1..length m] b)
```

We have introduced a new kind of node, (com n e), to represent (super-)combinators. (Com n e) represents a combinator with n arguments, which are referred to be the numbers 1..n in the body of the combinator. The combinator is constructed by the function mkcom, which used another version of subst to replace the maximal free expressions in the body of the combinator by integers. This version takes a list of expressions to replace (m) and a list of replacement values ([1..length m]). The function mkapp constructs an application of the new combinator to the maximal free expressions.

It only remains to define the function mfes, which identifies the maximal free expressions. This is easily done using an auxiliary function level which finds the maximum (most local) identifier in an expression. Then

```
mfes n e = [e], if level e<n
                (that is, if the most local identifier in e is more
                  global than n)
mfes n (app a b) = union(mfes n a)(mfes n b), otherwise
mfes n e = [], otherwise
```

The first equation says that if an expression contains only identifiers more global than n then it is itself a free expression. The second says that applications which are not free may yet have sub-expressions which are. Union is used so that the same

expression is not abstracted out more than once, and in effect performs common-sub-expression optimisation. The third equation says that other nodes (identifiers and combinators) have no sub-expressions which might be free. The last function we need, level, can be defined by

```
level (id n) = n
level (app a b) = max (level a) (level b)
level (com n b) = 0
```

Level is never applied to lam nodes and so does not need to be defined for them. A combinator is taken to be level 0 - that is, more global than any identifier, or effectively a constant. This is almost an efficient program. Strangely enough, mfes only visits each node a constant number of times [2]. However, level may be applied to each node many times. In fact, much better results are obtained if the mfes are sorted into increasing order of level before being substituted for [8], and this requires still more calls of level. If level is memoised then the algorithm presented here works well.

In fact, it is hard to write an efficient compiler to super-combinators without the aid of memo-functions. Other compilers (in imperative, functional and logic languages) appear in [8]; all are much more complex than this one. The functional version combines level, mfes, abstract and trans into one function in order to avoid repeated computation, but this is difficult because the patterns of recursion in each function differ. The resulting function is much larger than the sum of the sizes of the individual functions. Johnsson's lambda-lifting is a closely related problem: his compiler works in a similar way and is almost as complex [11]. The alternative solution of storing level information in the nodes before compilation is hard to apply because the easiest way to find the level of a lambda expression is to compile it.

4.6 **Summary**

In this section we have shown that lazy memo-functions can be used to make certain modular but inefficient algorithms efficient, where the alternatives destroy the program's structure in one way or another. We have argued in [9] that the most important advantage of functional languages is that they provide new forms of "glue" for combining solutions to sub-problems, thereby allowing problems to be decomposed in new ways and increasing modularity. These new glues are higher-order functions and lazy evaluation. Memo-functions are a third kind of glue in this sense.

5. **Hashing CONS and Full Memo-functions**

When we defined lazy memo-functions we relaxed the requirement that a memo-function avoid recomputation when applied to equal arguments, requiring it only to avoid recomputation when applied to identical ones. If "full" memo-functions (that is, the original kind) are really required they can be defined in terms of lazy ones.

This is done using a "hashing cons" [18]. A hashing cons (hcons) is like cons, but does not allocate a new cell if one already exists with an identical head and tail (that was allocated by hcons). Hcons is easily defined by

```
memo hcons a b = a:b
```

Using hcons, we can define a function that makes a unique copy of a structure.

```
unique (a:b) = hcons (unique a) (unique b)
unique x = x, otherwise          (x is atomic)
```

If a and b are two equal structures, then (unique a) and (unique b) are identical. This is clearly true for atoms, and therefore it is true for any structure built from atoms using cons by structural induction. Now, to make a function f into a full memo-function it is only necessary to apply unique to its arguments.

```
f x = g (unique x)
memo g x = ... body of f ...
```

When f is applied to equal arguments unique makes them identical, and then the lazy memo-function g avoids recomputation of the result. Of course, full memo-functions defined in this way suffer from all the same disadvantages as full memo-functions defined in any other way - they are relatively inefficient, completely strict, and the memo-tables keep growing.

6. **Implementation Issues**

Lazy memo-functions can be implemented by maintaining a memo-table of previous arguments and results. Since the test for re-occurrence of an argument is a pointer comparison rather than a recursive equality test the memo-table can be organised more efficiently than the memo-table for a full memo-function. It might be organised as a hash table or as a binary tree, for example. We won't go into such details here, but we will examine the problem of ever growing memo-tables.

In [13] Keller and Sleep give a comprehensive discussion of the implementation of full memo-functions. The most serious problem they raise is that memo-tables grow constantly - the more a memo-function is used, the more arguments and results are stored in its memo-table. Unless this growth is checked the memo-tables may become so large that *no space* is left for the rest of the computation. Keller and Sleep propose various caching strategies for deleting entries from memo-tables. Hilden compares strategies experimentally, for a particular function [6]. Unfortunately, any such strategy must be ad hoc, since one can never know for certain that a function will never be applied to a particular argument again.

Are lazy memo-functions any better? There are two ways in which they can help with this problem. Firstly, in many of the examples we discussed above the programmer knows when the memo-table should be emptied. For example, when using map with a cyclic argument it is vital that nothing should be deleted from the memo-table while a particular cyclic list is being mapped, for if it were a genuinely infinite result might be computed. However, once the mapping is over all entries for that cyclic list can be deleted. The programmer does not care whether or not he gets an identical result if he maps the same function over the same list again. He only cares that he gets a cyclic result each time he does it. It is sufficient to create a local memo-

table for each "top-level" application of map, which can be thrown away in its entirety when that application of map is completed. The programmer can elicit this behaviour by declaring a recursive memo-function local to map as follows:

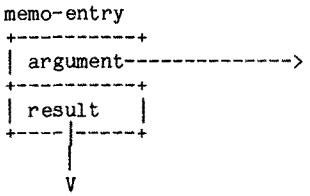
```
map f l = m l
    where memo m [] = []
          memo m (a:x) = f a:m x
```

With this definition a new memo-function with its own memo-table is created each time map is applied, and it exists just long enough to copy the cyclic argument, after which it is deleted in the normal way by the garbage collector. By localising memo-functions in this way the programmer can go a long way towards solving the problem himself.

Of course, there remain memo-functions which cannot be localised and so have long enough lifetimes to accumulate gigantic memo-tables. Most of the memo-functions in section 4 are of this type. Yet even here lazy memo-functions can make a contribution. Although it is impossible to predict that a function will never again be applied to an argument equal to a previous one, it is easy to predict that a function will never again be applied to an identical one when that argument is deleted by the garbage collector. So the garbage collector can be used to delete entries from memo-tables when they are no longer required. There is a difficulty here. By definition, all arguments stored in accessible memo-tables are accessible, and so a garbage collector using the normal rule that all accessible structures are preserved would not delete them. It follows that references from memo-tables to stored arguments must not count. The problem lies with references from memo-tables to stored results. If the corresponding argument is preserved then the result must also be preserved, since it must be returned should the memo-function be applied to the argument. On the other hand, if the argument is deleted then the result can be too. Since the garbage collector may find the memo-table before it has reached some of the stored arguments, it cannot know when dealing with the memo-table which of the stored results must be preserved. Consideration of memo-tables cannot be postponed to the end of garbage collection because some objects (and some stored arguments) may only be accessible via stored results of other stored arguments. We shall show how this problem can be resolved in mark-scan and copying collectors. The technique we use was first used by Friedman and Wise to delete entries from a scatter table [5].

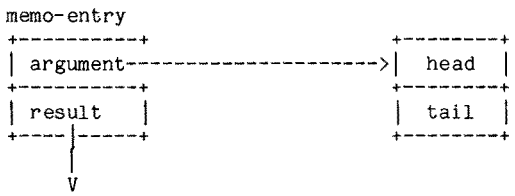
6.1 **Mark-scan Garbage Collection**

In the case of a mark-scan collector, we assume that all cells are the same size and have room for two pointers and some type bits. We assume that memo-table entries are stored in a cell of recognisable type

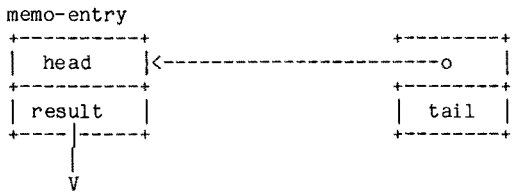


We assume that a memo-table is a list (or perhaps a binary tree) of memo-entries, and that the root cell is also of a recognisable type.

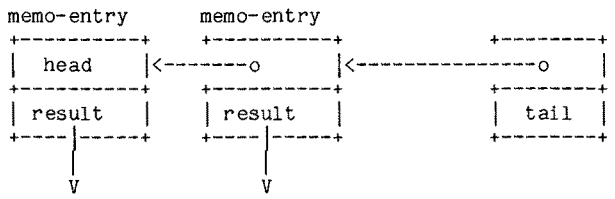
During the mark phase, when the garbage collector finds a memo-table it marks it, but marks the memo-entries only if the argument is already marked or is atomic. In these cases it also marks the result. If the argument is not yet marked, we have the structure



The garbage collector reverses the pointers from the memo-entry to the argument, producing the structure



If several memo-entries (in different memo-tables) refer to the same unmarked argument this results in a list of unmarked memo-entries hanging from the head of the argument.



When the garbage collector marks a cell whose head refers to a memo-entry it restores all these pointers to their original state and marks the memo-entries and their results. (The pointer stored in the head of the argument can be tagged specially if a bit is available so that no indirection is necessary to test whether it refers to a memo-entry). At the end of the mark phase all memo-entries, arguments and results that are still required have been marked. Memo-entries which are not required contain corrupted information.

The sweep phase proceeds as normal, except that when a marked memo-table is found, unmarked memo-entries are deleted from it and marked cells in the structure of the memo-table that are no longer required are returned to the freelist.

Phil Wadler has pointed out that the garbage collector could also delete memo-entries whose result is not pointed to from elsewhere, even if the argument is. This could make the program less efficient, but cannot otherwise affect its behaviour. It cannot even affect the results of future identity tests, because if the memo-entry is recomputed then the result can never be tested for identity against the previous result. We have not pursued this idea any further, as it would require a non-trivial modification to the algorithm.

6.2 **Copying Garbage Collection**

Copying garbage collectors [1] use two heaps, only one of which is normally active. When the active heap becomes full all accessible structures are copied onto the inactive heap, which then becomes the active one. It is often used in virtual memory implementations because the inactive heap costs nothing.

This kind of garbage collector can cope with variable sized nodes with no difficulty, so we assume that memo-tables are stored as contiguous vectors organised as hash tables. Each memo-entry consists of a pointer to the argument, a pointer to the result, and a pointer to the beginning of the memo-table. (This last pointer is present so that a pointer to a memo-entry also identifies a memo-table. It may not require a whole word on some machines since it can be stored as a relatively small offset). Since the copying process changes the addresses at which objects are stored memo-tables must be completely reorganised during garbage collection.

When a memo-table is copied to the new heap, all the entries in the new table are cleared and the address of the copy is stored in the header of the old one. Any entries whose arguments have already been copied are then inserted into the new table, and the corresponding results are copied. Entries whose arguments have not been copied have their pointers reversed, as in the mark-scan collector. The reversed pointer points at the memo-entry, not at the start of the memo-table. If one of these arguments is later copied to the new heap then the pointers are restored and the entry is inserted into the new memo-table (which can be found by following the pointer in the entry to the start of the old memo-table, and from the header of the old table to the new one). At the end of the garbage collection all accessible memo-tables have been reorganised and copied into the new heap, and all inaccessible arguments and results deleted.

This method neutralises one of the advantages of copying garbage collectors, that the new heap need only be accessed sequentially. Random access is required to the new memo-tables. This disadvantage can be minimised by storing memo-tables in a separate area of heap, so that the bulk of the new heap is still accessed sequentially. For example, each heap might be arranged with normal data stored at low addresses and memo-tables stored at high addresses, so that a garbage collection occurs when the two meet.

A representation with reversed pointers proved very useful in each garbage collection algorithm. This suggests yet another implementation of memo-functions, in which the memo-tables are always stored in the reversed form as "property lists" attached to nodes. Peyton-Jones has proposed such a representation as a way of "memoising the

data" rather than the function [16]. This has the advantage that memo-tables are likely to be small (perhaps just one or two entries), making memo-lookup very fast. The problems of garbage collection are eased because as soon as a data-structure is no longer required all its memo-entries automatically disappear. There is still a difficulty, namely that of deleting memo-entries when the memo-function referred to is no longer required, but this is less severe unless many different memo-functions are created dynamically (this is true of many of the examples above). A more serious disadvantage is that it is hard to see how functions of many arguments could be memo-ised in this way.

Of course, no garbage collection mechanism can delete entries with atomic arguments from memo-tables, because an atom (such as "2") cannot be deleted in any sense. Therefore certain lazy memo-functions (such as hashing cons) still accumulate larger and larger memo-tables. Nevertheless, the problem is much reduced - it now arises only when one insists on using lazy memo-functions to implement full ones. In examples such as those in section 4 these techniques should prove very effective.

7. Conclusion

We introduced a slight variation on the old idea of a memo-function. Our memo-functions combine well with lazy evaluation, and so we call them "lazy memo-functions". Lazy memo-functions increase the expressive power of a functional language, allowing cyclic structures to be treated as first class citizens, and allowing many modular but unreasonably inefficient algorithms to be used directly. Bird and Turner have observed that such algorithms are often derived during the development of a program. Lazy memo-functions can speed up program development by allowing it to stop at that point. The stages which are omitted thereby can destroy the structure of a program and render it obscure. Lazy memo-functions can be implemented efficiently and reduce the effect of "expanding memo-tables" considerably.

We have not discussed methods for reasoning about memo-functions. Although memoisation (of semi-strict functions) cannot compromise correctness, it does affect space- and time-efficiency. This adds an extra dimension to the analysis of efficiency which could make it much more difficult. In this paper we have reasoned very informally about the efficiency of such programs, but a more formal treatment must await future work. This is probably the most serious disadvantage of lazy memo-functions.

Acknowledgements

I am very grateful to Richard Bird, who provided the original motivation for this work; and also to Phil Wadler, Tony Hoare, Simon Peyton-Jones, and David Wise, all of whom made useful comments. I must also thank the United Kingdom Science and Engineering Research Council for supporting me with a European Research Fellowship while this work was carried out, and the members of the Programming Methodology Group at Chalmers University Göteborg for providing a stimulating environment.

References

[1] H. Baker, "List Processing in Real-time on a Serial Computer", Communications of the ACM, Vol. 21 no. 4, pp. 280-294 (April 1978).

[2] R. S. Bird, "Super-combinator Compilers: A Transformational Approach", Oxford University (1984).

[3] R. S. Bird, "Private communication", 1984.

[4] M. Coutts-Smith, "Expert Systems", MSc dissertation, Oxford University (1984).

[5] D. P. Friedman and D. S. Wise, "Garbage collecting a heap that includes a scatter table", Information Processing Letters, Vol. 5 no. 6, pp. 161-164 (December 1976).

[6] J. Hilden, "Elimination of recursive calls using a small table of 'randomly' selected function values", BIT, Vol. 16 no. 1, pp. 60-73 (1976).

[7] J. Hughes, "Super Combinators - A New Implementation Method for Applicative Languages", pp. 1-10 in Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh (1982).

[8] R. J. M. Hughes, "The Design and Implementation of Programming Languages", Programming Research Group Technical Monograph PRG-40, Oxford University (1983), Thesis.

[9] R.J.M. Hughes, "Why Functional Programming Matters", Memo 40, Programming Methodology Group, Chalmers University of Technology, Göteborg (1984).

[10] T. Johnsson, "Efficient Compilation of Lazy Evaluation", pp. 58-69 in Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, Montreal (June 1984).

[11] T. Johnsson, "Lambda Lifting", Memo 41, Programming Methodology Group, Chalmers University of Technology, Göteborg (1984).

[12] R. M. Keller and G. Lindstrom, "Parallelism in Functional Programming through Applicative Loops", University of Utah, Salt Lake City ().

[13] R.M. Keller and M.R. Sleep, "Applicative caching: Programmer control of object sharing and lifetime in distributed implementations of applicative languages", pp. 131-140 in Proceedings of the ACM Conference on Functional Languages and Computer Architecture, Wentworth (1981).

[14] D. Michie, "'Memo' functions and machine learning", Nature, No. 218, pp. 19-22 (April 1968).

[15] F. L. Morris, "On list structures and their use in the programming of unification", School of Computer and Information Science, Syracuse University (August 1978).

[16] S. L. Peyton-Jones, "Private communication", October 1984.

[17] E. H. Shortliffe, MYCIN: Computer-based Medical Consultations, Elsevier, New York (1976), based on a PhD thesis, Stanford University, 1974.

[18] Spitzen and Levitt, Communications of the ACM, Vol. 21 no. 12, pp. 1064-1075 (December 1978).

[19] D. A. Turner, "A New Implementation Technique for Applicative Languages", Software - Practice and Experience, Vol. 9, pp. 31-49 (1979).

[20] D. A. Turner, "The Semantic Elegance of Applicative Languages" in Proceedings 1981 Conference on Functional Languages and Computer Architecture, Wentworth-by-the-Sea, Portsmouth, New Hampshire (October 1981).

[21] D. A. Turner, "Recursion Equations as a Programming Language", pp. 1-10 in Functional Programming and its Applications, ed. D. A. Turner, Cambridge University Press, Cambridge (1982).

[22] D. S. Wise, Abacus, Vol. 2 no. 2, pp. 20-32 (Winter 1985).