Using Automatic Memoization as a Software Engineering Tool in Real-World AI Systems

James Mayfield Tim Finin

Computer Science Department
University of Maryland Baltimore County
Baltimore, MD 21228–5398 USA

Marty Hall

Eisenhower Research Center, JHU/APL
Johns Hopkins Rd.
Laurel, MD 20723 USA

Abstract

Memo functions and memoization are well-known concepts in AI programming. They have been discussed since the Sixties and are often used as examples in introductory programming texts. However, the automation of memoization as a practical software engineering tool for AI systems has not received a detailed treatment. This paper describes how automatic memoization can be made viable on a large scale. It points out advantages and uses of automatic memoization not previously described, identifies the components of an automatic memoization facility, enumerates potential memoization failures, and presents a publicly available memoization package (CLAMP) for the Lisp programming language. Experience in applying these techniques in the development of a large planning system is briefly discussed.

1 Introduction

Memo functions and memoization are well known concepts in AI programming. They have been discussed since the Sixties and are often used as examples in introductory programming texts [15, 12, 13]. The term "memoization" is derived from the term "memo function," which was coined by Donald Michie [9]. It refers to the tabulation of the results of a set of calculations to avoid repeating those calculations. *Automatic* memoization refers to a method by which an ordinary function can be changed mechanically into one that memoizes or caches its results. We place the decision about which functions to memoize in the hands of the user; this contrasts with the approach of Mostow and Cohen [10], which tries to automatically infer which functions should be memoized. These two approaches to automatic memoization are not incompatible, although as Mostow and Cohen point out, the latter approach is not a practical tool.

Memoization is particularly apt for AI applications. Rapid prototyping is a hallmark of AI programming. Automatic memoization allows the programmer to write certain functions without regard to efficiency, while being assured of reasonable performance. By allowing the programmer to avoide these efficiency concerns at the outset, automatic memoization facilitates the kind of exploratory programming used in the development of most AI systems.

The principle of memoization and examples of its use in areas as varied as logic programming [16, 6, 2], functional programming [4], and natural language parsing [11] have been described in the literature. In all of the cases that we have reviewed, the use of memoization was either built into in a special purpose computing engine (*e.g.*, for rule–based deduction, or CFG parsing), or treated in a cursory way as an example rather than taken seriously as a practical technique. The automation of function memoization as a practical software engineering technique under human control has never received a detailed treatment. In this paper, we report on our experience in developing CLAMP, a practical automatic memoization package for Common Lisp, and its use in a large–scale AI project.

By means of illustration, consider the divided–difference algorithm `dd`, which is shown in pseudo–code in Figure 1. This algorithm is used to determine coefficients of interpolated polynomials. The algorithm, which is a standard one in numerical methods, is taken directly from *Numerical Mathematics and Computing* [1]. The application itself is not particularly important; it is the run–time behavior of the algorithm that is of interest to us here. The call tree for the divided difference algorithm `dd` forms a network rather than a tree; thus, a recursive call to `dd` with particular arguments may be repeated many times during a single computation. For example, a call to `dd` with `low`=1 and `high`=10 will generate one recursive call with `low`=2 and `high`=10 and another recursive call with `low`=1 and `high`=9; each of these will in turn make a recursive call with `low`=2 and `high`=9. Memoization of `dd` causes each calculation to be performed only once and stored in a table; thereafter, each repeated recursive call is replaced by a table lookup of the appropriate value.

Figure 2 compares the performance of memoized and unmemoized versions of a Lisp[1] implementation of the divided difference algorithm, using $f(n) = \pi\cos(n)$ and the first n natural numbers as arguments. Since a single function call on n points generates two recursive calls on $n-1$ points, the unmemoized ver-

[1]Throughout this paper, we use the name Lisp to refer to the language Common Lisp, as described in Steele [14].

1043-0989/95 $4.00 © 1995 IEEE

```
; divided-difference algorithm
dd(points: array,
   low: array-index,
   high: array-index,
   fn: function)
begin
  if low = high then
    return fn(points[low])
  else return (dd(points, low+1, high, fn) -
               dd(points, low, high-1, fn)) /
               (points[high] - points[low])
end
```

Figure 1: The divided difference algorithm for determining coefficients of interpolated polynomials can be elegantly written using recursion and made efficient through the use of automatic memoization.

n	Unmemoized	Memoized *(first run)*	Memoized *(subsequent runs)*
15	11	0.18	0.0006
16	22	0.21	0.0006
17	43	0.22	0.0006
18	87	0.28	0.0007
19	173	0.4	0.0007
100	Centuries	25.0	0.002

Figure 2: This table shows the benefits of memoization on a Lisp implementation of dd. The time complexity is reduced from $\Theta(2^n)$ to $\Theta(n^2)$ for initial runs and to near-constant time on subsequent ones.

sion has $\Theta(2^n)$ time complexity. After memoization, the first invocation requires $\Theta(n^2)$ time, since no subsequence of points is calculated more than once, and there are $(n^2 + n)/2$ subsequences. Subsequent invocations take near-constant time.

This algorithm for divided difference is typical of a large class of algorithms which have very elegant recursive definitions which are simply unusable for most real problems. The conventional response to this situation is to manually rewrite the algorithm in a dynamic programming style. Of course, such manual rewriting of the code takes effort and involves the risk of introducing new bugs. An attractive alternative is to use an automatic memoization package to convert the elegant but inefficient recursive algorithm into a useful one. This is attractive, of course, only if such a package can address the practical problems faced in real application.

In the next section we describe the some of the aspects and uses of automatic memoization not previously described in the literature. Section three compares the use of automatic memoization to the alternative of rewriting code by hand. Section four describes the general components that should be present in any automatic memoization facility. Section five presents problems inherent in the use of memoization and the various ways to address them. Section six describes our experience in using automatic memoization in in the development of SMS [17], a decision support system that provides submarine crews with situational awareness and operational advice to reduce detectability.

2 Uses of Memoization

There are four main uses of automatic memoization. Two of these involve the avoidance of redundant calculation, first, within a single function invocation, and second, across invocations. The third use of automatic memoization is as a pre-calculation tool, while the fourth is as a timing and profiling tool. These are not conjectured uses but ones which we found to be effective in many situations in a large, real-world AI application. We discuss each of these uses in more detail in the following subsections.

2.1 Repetition within a Function Call

The most common use of memoization is to avoid the repetition of sub-calculations within a single function call. In the divided difference example presented above, there were many repeated recursive calls within a single function invocation. This type of repetition is common. For example, a simple recursive backtracking parser may parse the same constituent many times during a single parse; thus its performance is poor. Norvig [11] has shown that such an algorithm can obtain the performance of chart parsing [7] or of Earley's algorithm [3] through the application of memoization.

2.2 Repetition over Time

In a team programming environment different sections of a system, written by different programmers, may access the same function. Alternatively, in an interactive system the user may invoke calculations at different times that make use of some the same pieces. In these cases, there is no central routine which could manage the calling sequence to avoid repetition of the calculations. The only alternative to automatic memoization in such cases is to have the routine in question manage its own data structures to cache previous results.

2.3 Persistence

The preceding subsections showed that memoization can eliminate the repeated invocation of expensive calculations. These two applications of memoization are useful when it is feasible to perform the first invocation of a function at run-time. Memoization is also useful when even the first invocation is too expensive to perform at run-time.

Use of functions that are too expensive to calculate at run-time is usually done by building a special purpose data file, and storing in it the results of an off-line execution of the expensive routine. Then, the function in question is modified to access that file.

Automatic memoization provides a method to pre-calculate a function without the overhead of a hand-crafted solution. In such situations automatic memoization eliminates the need for the programmer to know which ranges of values are stored in the data file, and which must be calculated. To achieve persistence of an expensive function, the function is memoized and then run off-line on the cases of interest. The contents of the hash table are then saved to disk. The saved file is later used to seed the hash table for the function when it is reloaded.

There are two additional advantages of this type of memoization beyond providing the ability to pre-calculate a function. First, it allows the elimination of functions from the run-time system in cases where all possible inputs to the memoized function are pre-calculated. Second, the input values to the memoized function are determined automatically. That is, the programmer does not have to specify the range of possible inputs to the memoized function. In fact, this solution works even if the programmer has no idea which input values will be used.

2.4 Timing and Profiling

Finally, automatic memoization can also used as a profiling and timing tool. Many programming language systems provide a profiling facility whereby the user can see the time that a top-level function spends in various lower-level routines. This is important for directing optimization efforts. However, these profilers generally require significant overhead. For example, a fully-metered Lisp run on a Symbolics Lisp machine can take thirty times longer than an unmetered run. This does not include the time required to load the metering system. The expense of metering is worth the effort for important cases, and is a valuable software engineering tool. In smaller cases, however, automatic memoization provides a quick but rough method for determining which routines to optimize. Rather than running the fully metered system, users interactively memoize certain functions, then repeat the original test case twice. If the timing for the second case improves only by, for example, five percent, then for that test case, no amount of optimization in the routines in question will provide more than a five percent speedup. If on the other hand a great speedup is seen, then the memoized functions are good candidates for optimization.

3 Alternatives to Automatic Memoization

There are three alternatives to automatic memoization for eliminating repeated calculation: hand-crafted memoization, dynamic programming, and development of a new algorithm. First, memoization need not be automated. Memoizing a routine by hand could conceivably result in minor efficiency gains over automated memoization. Second, in some cases an ordering can be found for the calculations to be performed such that full memoization is not needed. For example, in Volume 2 (Seminumerical Algorithms) of his *The Art of Computer Programming* [8], Knuth presents a straightforward method for calculating divided differences in the proper order to get the same performance as the first invocation of the memoized version presented in the Introduction. Finally, a new algorithm for a given task can be sought that does not require repeated calculations.

Automatic memoization is not a substitute for finding the proper algorithm for a task. However, when the major benefit of the development of a new algorithm is a savings in repeated calculations, automatic memoization of an existing algorithm has several advantages. These advantages also recommend automatic memoization over the other approaches described above. They fall into three categories: quality of solution, ease of use, and additional uses of memoization.

3.1 Quality of Solution

Automatic memoization usually leads to short, clear implementations, because the code to implement the efficiency improvement does not appear in the body of the function being improved. Furthermore, if the function to be memoized has already been written and debugged, the use of automatic memoization does not risk the introduction of bugs into the function to the same degree that developing new code for the task does. This last point is especially important in the development of large, complex system where there is a natural reluctance to change routines that have already been tested and verified, especially if that will require changes in multiple places in the code. Furthermore, because it is simple to switch back and forth between the memoized and unmemoized versions, it is easy to compare the performances of the two versions.

3.2 Ease of Use

In most languages automatic memoization can be implemented so that it is simple to memoize and unmemoize functions. None of the alternatives to automatic memoization can boast such a light load on the programmer. Again, this ease of use depends in part on avoiding the requirement of writing, debugging and eventually maintaining new code. This ease of use is especially important in artificial intelligence applications, because the design of such applications tends to change rapidly and frequently.

3.3 Additional Uses

While the alternatives to automatic memoization mentioned above eliminate repeated calculations, they do not, in general, provide the other benefits of automatic memoization, *i.e.* persistence of cached values and usefulness as a timing and profiling tool. It would be possible, of course to build in some of these features such as a persistent cache mechanism. However, the automatic memoization approach requires us to do this only once—in the general memoization facility.

The result of these benefits is that automatic memoization offers significant practical advantages in building real systems. Hall and Mayfield [5] describe some of these advantages in more detail as they applied to the development of the SMS system.

```
(defun Hailstone (n)
  (+ 1  (cond ((= n 1) 0)
              ((evenp n) (Hailstone (/ n 2)))
              (t (Hailstone (+ 1 (* 3 n)))))))
```

Figure 3: The Hailstone function appears to be a good candidate for memoization, but it is not.

4 Components of an Automatic Memoization Facility

Based on our experience in developing and using CLAMP, we have identified characteristics that any automatic memoization facility should have. Automated memoization of some sort can be implemented in most languages. Languages that provide the user with an interactive run-time environment (such as Lisp, Dylan and ML) must have either run-time function name resolution or a patchable code segment to allow automated memoization. Languages that are strictly compiled (such as most implementations of C) must be augmented with some sort of preprocessor to allow automated memoization.

Three types of control functions make up a useful memoization package: memoization and unmemoization functions, cache control functions, and statistics reporting functions. We will discuss each in turn.

4.1 Memoization Control

First, the system must provide the user with a variety of methods to easily memoize and unmemoize functions. These methods should allow both absolute control over memoization (*e.g.* memoize a function, unmemoize a function, unmemoize all functions), and also, if possible, temporary memoization commands (*e.g.* treat a function as memoized while evaluating a particular expression, treat a function as unmemoized while evaluating a particular expression).

It is important to allow the programmer to experiment with the effects of memoization in the context of his or her program. It is not always obvious what impact memoizing a function will have. For example, at first glance, the Hailstone function shown in Figure 3 looks to be a good candidate for memoization, but experimentation shows no benefits from doing so.[2]

4.2 Cache Control

Second, the system must provide methods to control individual memoization caches, both for a single session and across sessions. Most important is the ability to explicitly clear a cache, thereby forcing all function return values to be re-calculated. Persistence is provided by allowing the user to instruct that a cache be saved to disk, and in a later session, to instruct that such a saved cache be read from disk. Ease of use concerns dictate that these methods be activated by referring to the name of the memoized function, and not to the name of a disk file or to some other internal data structure.

[2]Of course, some careful reasoning will also lead a programmer to the same conclusion, but not all programmers will have the time and experience to do this type of analysis.

4.3 Data Control

Finally, the memoization package should provide routines that report statistics collected while a memoized function is running. These statistics should include the number of times a memoized function is called, and the number of times that such invocations result in the return of a cached value. The user should also be allowed to reset these statistics in preparation for a new run.

5 Memoization Failures

A major advantage of automatic memoization is its transparency. However, an overly-transparent view can lead to problems. While some aspects of these memoization failures have been discussed in the literature (notably by Mostow and Cohen [10]), most have not. Instead, we learned them the hard way in using the evolving CLAMP system through the experiences of AI programmers using the facility over the course of several years in developing the SMS system.

The most common criticism of automatic memoization that we hear from programmers who have not used it is that the use of a technique that cannot guarantee correct results after its application is out of the question. This complaint is unfounded. As a counterexample, a technique that is widely-used by C programmers is function in-lining through macro expansion. Such macro expansion is not guaranteed to produce the correct results in all cases. However, because the programmer controls when macros are applied, the technique can be used profitably. It is for this reason that we advocate programmer control over the selection of functions for memoization. In the following subsections, we describe the potential pitfalls we have encountered in making decisions about which functions to memoize.

5.1 Cache Consistency

Memoization is used to avoid recalculation when a function is invoked with arguments it has seen previously. Direct changes to the function should result in the memoization cache being flushed automatically, as presumably it will then contain incorrect values. This situation is relatively easy to detect. More difficult to detect is the alteration of one or more of the sub-functions that the memoized function calls. There are several ways to alleviate this problem. The best method would be for each memoized function to keep a list of all functions in its call tree, and require that the cache be flushed when any of those entries is modified. This could not be done automatically at run-time without the use of special data structures in the unmemoized functions; this capability is unlikely to be adopted in most programs. In many cases, a proper use of modularity will indicate how far changes are likely to propagate. Since the user always has access to a list of the currently memoized functions, a warning to check the list whenever changes are made is often sufficient. This is a trade-off. One of the goals of an automatic memoization facility is to provide tools that are easy for the programmer to adopt. If using memoization requires invasive changes to unmemoized routines, this goal will be compromised.

However this problem is addressed within a single session, it is usually not a problem across sessions. Run-time changes to functions usually occur during development. If caches are not saved to disk, the memoized routines will once again calculate correct values the next time the system is loaded.

Caches that are saved to disk present a more serious problem. In some senses, such caches are no different from normal data files, and the same problems of outdated versions apply. No fully automated solution will guarantee consistency, short of a system builder that forces recalculation of all saved values whenever there are changes in any code on which the function depends. This is not an unreasonable option during development, since the memoization facility makes it easy to save entries. Off-line calculations can be performed at regular intervals. However, there is still an incentive to limit these calculations, since their time-consuming nature is what led to saving the tables to disk in the first place.

One way to limit the likelihood of inadvertent use of obsolete values semi-automatically is to periodically recalculate memoized values. First, the programmer specifies a percentage of the entries that are to be recalculated on loading the hash table, and/or a percentage of times where the memoized function will invoke the original function even though its arguments have been seen before. In each case, these recalculated values are compared to the original ones; a warning is given if the results are not identical. Load-time recalculation may not be possible, of course, since all the routines needed for a calculation may not be available when the function is defined. Similarly, run-time recalculation may not be desirable, even with a very low percentage specification, if the system is counting on a given level of performance. Even if both techniques are used, they do not guarantee that all obsolete values will be found. Thus, these techniques should be used as a complement to other software engineering techniques (such as explicit maintenance of dependencies), rather than as a replacement for them.

5.2 Non-Functions

Memoization only works for true functions, not procedures. That is, if a function's result is not completely and deterministically specified by its input parameters, using memoization will give incorrect results. The number of functions that can be memoized successfully will be increased by encouraging the use of a functional programming style throughout the system.

5.3 Modification of Results

Inherent in the idea of memoization is that data is stored, rather than calculated anew each time a function is called. Thus, memoized functions can return results that share structures, even if the unmemoized version always creates new structures. Even without memoization, operations that modify function results are dangerous from a software engineering perspective. A common problem is that such routines will work fine when first written, but will make subsequent modifications difficult. However, in some cases they can lead to efficiency gains; with care, programmers can use them to speed up the functions that can really benefit from their use. Unfortunately, the transparent view of memoization breaks down when used with such routines.

For instance, suppose that function **Raw-Data** returns a newly-created list of numbers. It is called by the function **Normalized-Data**, which destructively removes the maximum and minimum entries from the list before returning it. Prior to memoization, this might be perfectly safe. After memoizing **Raw-Data**, however, each subsequent retrieval of supposedly identical data values might in fact receive a shorter list. Avoiding this problem not only requires the user of memoization to know how the function to be memoized works, but also how it will be used by other functions. This is often a difficult task; an easier approach is to tighten the standards on when modifying operations are allowed, and to require increased documentation for those functions that truly need to use them.

5.4 Compiler Optimization of Recursion

Some compilers will optimize the code they output by converting tail-recursion to iteration. Such optimization eliminates the recursive function call, but not the work required to calculate the function's result. Since memoization relies on explicit function calls to activate its table-lookup, such optimization will bypass the memoization process. To avoid this problem, the compiler must be instructed not to eliminate tail-recursion in memoized routines. Compilers that do optimize tail-recursion usually provide an appropriate compiler directive, for use by the memoization machinery.

A more subtle optimization is sometimes made by compilers for languages that resolve function names at run-time. Such compilers will often bypass the name resolution process for direct recursion. When memoization is implemented by binding the memoized function to the original function name, this once again results in a circumvention of the memoization table-lookup. The function will still return correct results, but the computation savings provided by memoization will be lost. It is less common for a compiler to give the user explicit control over this kind of optimization.

Note that this problem can eliminate some, but not all, of the advantages of memoization. Although the results of the optimized-away *recursive* calls will not be cached, the results of the top-level calls will be cached. In the terms of the use categories described in Section 2, the benefits of *repetition within a function call* are lost but those due to *repetition over time* are not.

5.5 Recognizing Call Equivalence

Memoization is performed by doing an exact match on the argument list, using the Lisp function **equal** by default. If function **Foo** is defined as:

```
      (defun Foo (&key (Bar 2) (Baz 3) ...)
```

and is memoized, all of the following will be treated as distinct, even though the parameters have identical values in all cases:

```
(Foo)
(Foo :Bar 2)
(Foo :Bar 2 :Baz 3)
(Foo :Baz 3)
(Foo :Baz 3 :Bar 2)
```

Similarly, one can have counterintuitive results when the arguments are floating point numbers, forgetting that, for instance, 2 is not `equal` to 2.0, and 1.234567 is not `equal` to 1.23456, even though the function may treat them as identical. The solution adopted by the SMS program is to introduce "wrapper" functions that take keyword arguments, floating point numbers, *etc.*, canonicalize the arguments into some common form, then pass them on to an internal function that takes only required arguments in the standard format. It is this internal function that is then memoized.

5.6 Cache Value Representation

In the current system, the routines that save data to disk do so by printing the representation of the object using `format`, directing the output stream to a file. This means that Lisp objects whose print representation cannot be parsed by `read` cannot be saved to disk. Some objects such as CLOS instances and structures allow the definition of a custom print function, and this can sometimes be used to save them to disk. But this is not a general mechanism, and special-purpose code will need to be written in those cases.

6 Experience and Evaluation

The Signature Management System (SMS) is a decision aid for submarine crews that provides situational awareness and operational advice to help the ship reduce its overall detectability. It has been developed over the last five years under ARPA funding at the Johns Hopkins University Applied Physics Laboratory (JHU/APL). Outside of APL, team members have come primarily from industry, with eight corporations and two universities on the development team. The system combines a variety of representations including frames/objects, production rules, mathematical models, and procedural code. About 75% is written in Common Lisp, with the remainder in C; it runs on UNIX platforms.

The automatic memoization system was used in the SMS program by at least six Lisp developers from three different companies. "Permanent" memoization (*i.e.* uses other than profiling) remains in the released system in 25 places. However, use as a first-cut profiling tool was one of the most common uses of memoization. All four uses described in Section 2 were used extensively by multiple programmers.

6.1 SMS Magnetics Module

Figure 4 gives timing statistics for a magnetics module used in the Signature Management System, timed after various uses of memoization were put into effect. Ignoring the benefits when the user asks for the same display at different times (which is in fact quite common), Figure 4 gives a summary of the time benefits of memoization on the first time invocation of the top-level display. Times are in seconds, and are conservative approximations. Similar results were obtained with other modules.

aspect	Time (sec)	Speedup
unmemoized original	48	1.0
+ conventional optimization	36	1.33
+ repetitions over time	24	2.0
+ dynamic programming	2	24.0
+ saved memoization tables	0.001	48,000

Figure 4: These figures show the cumulative effects of the different aspects of automatic memoization on a magnetics module used in the Signature Management System.

6.2 SMS Detectability Planning Display

Given the diverse uses of memoization by various programmers on the SMS program, we attempted to estimate the overall contribution of memoization to the system. For instance, one of the displays used as an aid to planning submarine operations in the SMS system shows the predicted probability of detection of the submarine for various choices of heading and speed, drawn on a polar plot with the angle (theta) indicating heading (0 corresponding to due north), and the radius (r) corresponding to speed. Each (r,theta) pair (arc) in the display is coded with a color indicating the cumulative probability of detection for the sub if it were to operate at the indicated course and speed.

This display is used as a high-level tool in planning, and thus includes highly summarized information. It presents a single number for probability of detection which is a composite of all the potential detection methods or ***signatures***. The user frequently is interested in the contribution of individual signature components to this composite. Since the probability of detection of each component is memoized before it is combined into the composite, any component corresponding to a point on the display can be retrieved almost instantly. Taking advantage of this, the display of can be maintained ***with virtually no additional computation***.

Whenever the user moves the mouse over the composite detectability display, the corresponding speed and course for the point under the mouse is calculated. Then, the individual components are calculated, with their relative values shown in the bar charts. Due to the effects of memoization, the component values can be calculated and graphed as quickly as the user can move the mouse.

The system was run from this display in the default mode and then with all memoization turned off. The results, given in Figure 5 show a 631x improvement in speed, and a 4,822x improvement in the amount of temporary memory (garbage) allocated. Benchmarks are notoriously misleading, and in many places the code would have been written dramatically differently if memoization had not been available. Nevertheless,

version	time	bytes consed
unmemoized	2562.74 sec.	2,969,392,724
memoized	4.06 sec.	615,784

Figure 5: Dramatic improvements in execution time and consing were obtained in the overall SMS system.

the results are illuminating, especially since they represent improvements over the original baseline system. Because the computation of this summary display represents the final, high-level computation of the entire system it is a reasonable way to measure the contribution of the use of automatic memoization.

7 Conclusions

Automatic memoization is a powerful tool that allows many simple but inefficient algorithms to be made useful in practice. Beyond this basic advantage though, automatic memoization provides other significant advantages to the artificial intelligence programmer. These advantages include the ability to add persistence to a memoized function, and the ability to perform timing and profiling studies rapidly. These advantages far outweigh the potential pitfalls of automatic memoization in artificial intelligence applications, because of the prevalence of a rapid prototyping approach in such projects.

Source code for the CLAMP system is available via anonymous FTP (ftp://ftp.cs.umbc.edu) or by email via a request to (hall@cs.umbc.edu). CLAMP is also available on the Internet Lisp archives at CMU, and is part of the CMU AI CD-ROM.

Acknowledgements. This work was supported in part by the Advanced Research Projects Agency under JHU/APL subcontract 605089-L. The authors thank V.J. Benokraitis (AAI Corporation), Lien T. Duong (AAI Corporation), J. Paul McNamee (AAI Corporation), Peter Norvig (Sun Microsystems), and David J. Scheerer (The Johns Hopkins Applied Physics Laboratory) for their helpful comments, both on the LISP implementation and on earlier versions of this paper. Thanks to John Aspinall (Symbolics) for suggesting the use of the `divided--difference` example.

References

[1] Ward Cheney and David Kincaid. *Numerical Mathematics and Computing.* Brooks/Cole, 1980.

[2] S.W. Dietrich. Extension tables: Memo relations in logic programming. In *Fourth International Symposium on Logic Programming*, pages 264-273, 1987.

[3] J. Earley. An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery*, 6(2):451-455, 1970.

[4] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[5] Marty Hall and James Mayfield. Improving the performance of AI software: Payoffs and pitfalls in using automatic memoization. In *Proceedings of the Sixth International Symposium on Artificial Intelligence*, pages 178-184. Megabyte, September 1993.

[6] B. Hoffmann. Term rewriting with sharing and memoization. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming: Proc. of the Third International Conference*, pages 128-142. Springer, Berlin, Heidelberg, 1992.

[7] M. Kay. Algorithm schemata and data structures in syntactic processing. In *Proceedings of the Symposium on Text Processing.* Nobel Academy, 1980.

[8] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1969.

[9] Donald Michie. "memo" functions and machine learning. *Nature*, 218(1):19-22, April 1968.

[10] Jack Mostow and Donald Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 165-172. Morgan Kaufmann Publishers, Inc., 1985.

[11] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91-98, 1991.

[12] Peter Norvig. *Paradigms of AI Programming: Case Studies in Common LISP.* Morgan Kaufmann, 1992.

[13] F. C. N. Pereira and S. M. Shieber. *Prolog and Natural-Language Analysis.* csli, Stanford, CA, 1987.

[14] Guy L. Jr. Steele. *Common Lisp: The Language.* Digital Press, second edition, 1990.

[15] Gerry Sussman and Hal Abelson. *The Structure and Interpretation of Computer Programs.* MIT Press, 1983.

[16] David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93-111, 1992.

[17] David Wenstrand, H. Lee Dantzler, Marty Hall, David Scheerer, Charles Sinex, and David Zaret. A multiple knowledge base approach to submarine stealth monitoring and planning. In *Proceedings of the DARPA Associate Technology Symposium*, June 1991.