Techniques for Automatic Memoization with Applications to Context-Free Parsing

Peter Norvig
Computer Science Division
University of California
Berkeley, CA 94720, USA
appeared in Comp. Linguistics 17 (1), March, 1991

Abstract

It is shown that a process similar to Earley's algorithm can be generated by a simple top-down backtracking parser, when augmented by automatic memoization. The memoized parser has the same complexity as Earley's algorithm, but parses constituents in a different order. Techniques for deriving memo functions are described, with a complete implementation in Common Lisp, and an outline of a macro-based approach for other languages.

1 Memoization

The term *memoization* was coined by Donald Michie [8] in 1968 to refer to the process by which a function is made to automatically remember the results of previous computations. The idea has become more popular in recent years with the rise of functional languages; Field and Harrison [4] devote a whole chapter to it. The basic idea is just to keep a table of previously computed input/result pairs. In Common Lisp one could write[1]:

```
(defun memo (fn)
  "Return a memo-function of fn."
  (let ((table (make-hash-table)))
    #'(lambda (x)
        (multiple-value-bind (val found)
            (gethash x table)
          (if found
              val
              (setf (gethash x table) (funcall fn x)))))))
```

(For those familiar with Lisp but not Common Lisp, `gethash` returns two values, the stored entry in the table, and a boolean flag indicating if there is in fact an entry. The special form `multiple-value-bind` binds these two values to the symbols `val` and `found`. The special form `setf` is used here to update the table entry for `x`.)

In this simple implementation `fn` is required to take one argument and return one value, and arguments that are `eql` produce the same value. Below we will relax some of these restrictions. The problem with this approach is that we also want to memoize any recursive calls that `fn` may make. To use the canonical example, if we have:

[1] All examples are in Common Lisp, rather than in generic pseudo-code for two reasons: First, I want to stress that these techniques can actually be used in existing languages. Second, Common Lisp provides a rich set of primitives (such as `make-hash-table`) that would otherwise require lengthy explanations.

```
(defun fib (n)
  "Compute the Nth Fibonacci number."
  (if (<= n 1) n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

then the function `(memo (function fib))` will not have linear-order complexity, because recursive calls will go to the original function `fib`, not to the memoized version. One way to fix this problem is to assign the new memoized function to the name `fib`. Common Lisp is a good host language for this approach, because there are primitives for accessing and altering the global function name space. Consider the following:

```
(defun memoize (fn-name)
  "Replace fn-name's global definition with a memoized version."
  (setf (symbol-function fn-name) (memo (symbol-function fn-name))))
```

When passed a symbol which names a function, `memoize` changes the global definition of the function to a memo-function. Thus, any recursive calls will go first to the memo-function, rather than to the original function. This is just what we want; all we have to say is `(memoize 'fib)` to transform `fib` from an exponential- to a linear-order algorithm.

To make sure that the memoization step isn't left out during program development, some programmers may prefer to write code like this:

```
(memoize
  (defun f (x) ...))
```

Or even like this:

```
(defmacro defun-memo (fn args &body body)
  "Define a memoized function."
  `(memoize (defun ,fn ,args . ,body)))

(defun-memo f (x) ...)
```

Both of these approaches rely on the fact that `defun` returns the name of the function defined. Note that the following will *not* work,

```
(labels ((fib (n)
           (if (<= n 1) n
               (+ (fib (- n 1) (fib (- n 2)))))))
  (memoize 'fib)
  (fib 100))
```

because `memoize` affects only the global function binding, not the lexical (local) definition.

The version of `memo` presented above suffers from a serious limitation—the function to be memoized can only have one argument. In the revised definition given below, the function can take any number of arguments, and the indexing can be on any function of the arguments. When there is only argument, the default key function, `first`, is appropriate. To hash on all the arguments, use `identity` as the key.

```
(defun memo (fn &key (key #'first) (test #'eql) name)
  "Return a memo-function of fn."
  (let ((table (make-hash-table :test test)))
    (setf (get name 'memo) table)
    #'(lambda (&rest args)
        (let ((k (funcall key args)))
          (multiple-value-bind (val found)
              (gethash k table)
            (if found val
                (setf (gethash k table) (apply fn args))))))))
```

```
(defun memoize (fn-name &key (key #'first) (test #'eql))
  "Replace fn-name's global definition with a memoized version."
  (setf (symbol-function fn-name)
        (memo (symbol-function fn-name)
              :name fn-name :key key :test test)))

(defun clear-memoize (fn-name)
  (clrhash (get fn-name 'memo)))
```

Also note that the hash table is stored away on the function name's property list, so that it can be inspected or cleared at will. The intent is that the user's program should clear the table when the results are likely to be out of the working set. We might also want a hash mechanism that caches only recent entries, and discards earlier ones, as originally suggested by Michie [8].

The user also has the responsibility of choosing the appropriate hashing function. By choosing `eql` instead of `equal` hashing, for example, hashing overhead will be reduced, but computation will be duplicated for `equal` lists. A compromise is to use `eql` hashing in conjunction with unique or canonical lists [10]. `eql` hashing has the additional advantage of allowing infinite circular lists, as discussed by Hughes [6].

2 Context-Free Parsing

All efficient algorithms for parsing context-free grammars make use of some kind of well-formed substring table. Earley's algorithm [3] is perhaps the best-known example. The algorithm builds up a vector of parse lists, where each entry in a parse list is an item—a production rule with one indicator showing how much of the right hand side has been parsed, and another saying where the parse started. Kay [7] introduced a similar data structure called a chart, along with a family of parsing algorithms that operate on the chart. Chart parsing is described in virtually all recent texts on natural language processing; for example Winograd [11] devotes 19 pages to the topic. It is part of the "folklore" on parsing that these algorithms can be thought of as tabular (i.e. memoized) versions of corresponding simpler algorithms. Earley himself [3] mentions it, and Shieber [9] gives a general abstract parsing strategy, and then proves that Earley's algorithm can be derived by suitable constraints on control of the strategy.

This paper's contribution is a concrete demonstration of just how direct the correspondance is between the simple and the efficient algorithm. We present a simple parser which, when memoized, performs the same calculations as Earley's algorithm. The core of the parser is only 15 lines of code:

```
(defun parse (tokens start-symbol)
  "Parse a list of tokens, return parse trees and remainders."
  (if (eq (first tokens) start-symbol)
      (list (make-parse :tree (first tokens) :rem (rest tokens)))
      (mapcan #'(lambda (rule)
                  (extend-parse (lhs rule) nil tokens (rhs rule)))
              (rules-for start-symbol))))


(defun extend-parse (lhs rhs rem needed)
  "Parse the remaining needed symbols."
  (if (null needed)
      (list (make-parse :tree (cons lhs rhs) :rem rem))
      (mapcan
        #'(lambda (p)
            (extend-parse lhs (append rhs (list (parse-tree p)))
                          (parse-rem p) (rest needed)))
        (parse rem (first needed)))))
```

This assumes that there are no left-recursive rules. The parser requires the following definitions:

```
(defstruct (parse) "A parse tree and a remainder." tree rem)

;; Trees (and rules) are of the form: (lhs . rhs)
(defun lhs (tree) (first tree))
(defun rhs (tree) (rest tree))


(defun rules-for (symbol)
  "Return a list of the rules with symbol on the left hand side."
  (remove symbol *grammar* :key #'lhs :test-not #'eql))
```

We now need to specify the memoization. In addition, since the function `parse` returns all valid parses of all prefixes of the input, we add the function `parser` which returns only parses of the complete input.

```
(memoize 'rules-for)
(memoize 'parse :test #'equal :key #'identity)


(defun parser (tokens start-symbol)
  "Return all complete parses of a list of tokens."
  (clear-memoize 'parse)
  (mapcar #'parse-tree
          (remove-if-not #'null (parse tokens start-symbol)
                         :key #'parse-rem)))
```

As an example, consider the following grammar, taken from page 321 of Aho and Ullman [2]. Here the Lisp form `(E    T + E)` corresponds to the grammar rule $E \to T + E$, where `+`, `*`, `[` and `]` are all terminal symbols in the grammar, and concatenation is implicit.

```
(defparameter *grammar*
  '((E    T + E)
    (E    T)
    (T    F * T)
    (T    F)
    (F    [ E ])
    (F    a)))
```

We can use this to obtain parses like the following:

```
> (parser '( [ a + a ] * a) 'E)
((E (T (F [ (E (T (F A)) + (E (T (F A)))) ])
       *
       (T (F A)))))
```

We do not include here a proof that the memoized `parse` does the same work as Earley's algorithm, but it is not too difficult to show. Interested readers can insert the following `format` statement as the first expression in `extend-parse`, and then run the example again. The resulting output, when sorted, matches exactly the parse lists shown on page 322 of Aho and Ullman [2].

```
(format t "~&~d ~a ->~{ ~a~} . ~{ ~a~}"
  (- 7 (length rem)) lhs
  (mapcar #'(lambda (x) (if (consp x) (first x) x)) rhs)
  needed)
```

The items come out in a different order because the Earley algorithm is strict left-to-right, while the memoized version is doing backtracking, but with the memoization eliminating all duplication of effort.

Unfortunately, the memoized parser as presented does not have the same asymptotic complexity as Earley's algorithm. The problem is that hashing is done on the complete argument list to `parse`: the start symbol and the list of remaining tokens. Hashing on this takes time proportional to the number of tokens, so the whole algorithm is $O(n^4)$ instead of $O(n^3)$. What we really want is a hash table that is a compromise between `equal` and `eql` hashing, one that takes keys that are lists, where each element of the list should be compared by `eql`. Such a table can be easily managed:

```
(defun put-multi-hash (keylist value hash-table)
  "Store a value in a multi-level hash table:
  one level for each element of keylist"
  (if (= (length keylist) 1)
      (setf (gethash (first keylist) hash-table) value)
      (let ((table1 (or (gethash (first keylist) hash-table)
                        (setf (gethash (first keylist) hash-table)
                              (make-hash-table)))))
        (put-multi-hash (rest keylist) value table1))))

(defun get-multi-hash (keylist hash-table)
  "Fetch a value from a multi-level hash table:
  one level for each element of keylist"
  (if (= (length keylist) 1)
      (gethash (first keylist) hash-table)
      (let ((table1 (or (gethash (first keylist) hash-table)
                        (setf (gethash (first keylist) hash-table)
                              (make-hash-table)))))
        (get-multi-hash (rest keylist) table1))))
```

Now to use these multi-level hash tables, we need another version of `memo`, one that gives us the flexibility to specify the `get` and `put` functions:

```
(defun memo (fn &key name (maker #'make-hash-table)
             (getter #'gethash) (putter #'puthash))
  "Return a memo-function of fn."
  (let ((table (funcall maker)))
    (setf (get name 'memo) table)
    #'(lambda (&rest args)
        (multiple-value-bind (val found)
            (funcall getter args table)
          (if found val
              (funcall putter args (apply fn args) table))))))


(defun memoize (fn-name &rest memo-keys)
  "Replace fn-name's global definition with a memoized version."
  (setf (symbol-function fn-name)
        (apply #'memo (symbol-function fn-name)
               :name fn-name memo-keys)))
```

Finally, we can get an $O(n^3)$ parser by saying:

```
(memoize 'parse :getter #'get-multi-hash :putter #'put-multi-hash)
```

3 Memoizing in Other Languages

In the Scheme dialect of Lisp, the programmer would have to write `(set! fib (memo fib))` instead of `(memoize 'fib)`. This is because Scheme lacks `symbol-value` and `set-symbol-value!` functions, and has nothing to do with the fact that Scheme has a single name space for functions and variables. It is also possible to write the memoized function all in one step:

```
(define fib
  (memo (lambda (n)
          (if (<= n 1) n
              (+ (fib (- n 1) (fib (- n 2))))))))
```

This is the approach taken by Abelson and Sussman [1] (see exercise 3.27).

In a Scheme implementation with macros, we could of course define `memoize` or `define-memo` as macros, thereby achieving the same result as in Common Lisp. In fact, the macro solution even works in languages without first-class functional objects, if a sufficiently powerful macro facility is available, or if the language's parser can be modified. For example, one could add the keyword `memo` to Pascal, and write the following:

```
memo function fib(n:integer) : integer;
begin
  if n <= 1 then
      fib := n
  else fib := fib(n-1) + fib(n-2);
end;
```

and have it parsed as if it were:

```
function fib(n:integer) : integer;
begin
   if not inTable(n) then
      if n <= 1 then
         addToTable(n,n)
      else addToTable(n,fib(n-1) + fib(n-2));
   fib := tableValue(n);
end;
```

Texts such as Field and Harrison [4] assume that memoization is done by a built-in feature of the language that in effect implements this kind of source-to-source transformation. This paper shows that memoization can also be done completely within the language—provided the language has either (a) a macro facility or (b) both higher-order functions and a way to set function bindings.

4 Conclusion

What has been gained by memoization? The memoized algorithms presented here are of the same order of complexity as the exisiting Early algorithm and linear-recursive Fibonacci function. In fact, unless tables are implemented very carefully, the memoized algorithms perform worse by a constant factor.

However, note that Earley's algorithm is not the best for all parsing applications. One might prefer a bottom-up parser, or a deterministic parser. The point is that experimenting with alternate parsing strategies is easier when starting from a simple 15-line program, rather than a multi-page parser which maintains complex explicit data structures.

Memoization supports an incremental, lazy-evaluation-like style that can make the overall design simpler. For example, note that besides `parse`, we also memoized `rules-for`. This has the effect of building an inverted index of the grammar rules, implicitly and incrementally. The programmer is freed from having to remember to declare, allocate and initialize a rule table, but still gets the benefit of efficient access to the rules. This is the ultimate data abstraction: the programmer is shielded not only from the details of the implementation of the table, but even from the existence of a table at all.

For twenty years, algorithms like Earley's (and more recently chart parsing) have been treated as special techniques, worthy of special attention. This paper has shown that the maintenance of well-formed substring tables or charts can be seen as a special case of a more general technique: memoization. Furthermore, we have shown that Common Lisp, with its mutable function name space, is an especially congenial host language for memoization, and that other languages can be hosts with other approaches. It is our hope that programmers will adopt automatic techniques like memoization where appropriate, and concentrate their data-creation and algorithm-optimization efforts on truly special cases.

5 Acknowledgment

This work has been supported by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

References

[1] Abelson, H., and G.J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[2] Aho, A. V., and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol 1: Parsing* (Prentice-Hall, Englewood Cliffs, NJ, 1972).

[3] Earley, J., An efficient context-free parsing algorithm, *CACM* **6** (2) (1970) 451-455. Reprinted in [5].

[4] Field, A.J., and Harrison, P.G., *Functional Programming* (Addison-Wesley, Reading, MA, 1988)

[5] Grosz, B. J., and K. Sparck-Jones, and B. L. Webber, *Readings in Natural Language Processing* (Morgan Kaufman, Los Altos, CA, 1986).

[6] Hughes, R.J.M. 'Lazy memo functions' In *Proc. Conference on Functional Programming and Computer Architecture*, Nancy, 129-146 Springer-Verlag, 1985.

[7] Kay, M., Algorithm schemata and data structures in syntactic processing, in *Proceedings of the Symposium on Text Processing*, Nobel Academy, (1980). Reprinted in [5].

[8] Michie, D. "Memo" functions and machine learning, Nature, 218, (1968) 19-22.

[9] Sheiber, Stuart M., Parsing and type inference for natural and computer languages, SRI International Technical Note 460, (1989).

[10] Szolovits, P. and W.A. Martin, 'Brand X: Lisp support for semantic networks,' in *Proc. of the 7th IJCAI*, Vancouver B.C. (1981).

[11] Winograd, T., *Language as a Cognitive Process* (Addison-Wesley, Reading, MA, 1983).