Using memoization to achieve polynomial complexity of purely functional executable specifications of non-deterministic top-down parsers

R. A. Frost[*], *School of Computer Science, University of Windsor, Windsor, Ontario, Canada N9B 3P4*

Introduction

Norvig [1991] has shown that a process with properties that are similar to those of Earley's algorithm can be generated by a simple top-down backtracking parser when augmented by memoization.

Unfortunately, Norvig's approach cannot be implemented directly in pure functional programming languages such as Miranda, Haskell, LML, etc, owing to the fact that such languages do not provide any form of mutable object. This restriction is a necessary condition for referential transparency, a property of programs that simplifies reasoning about them and which is widely regarded as one of the main advantages of the pure functional programming style.

In this note, we show how Norvig's approach can be adapted and used to achieve polynomial complexity of non-deterministic top-down backtracking parsers that are implemented as executable specifications in pure functional programming languages. The technique that we present uses a slight variation of an approach to implementing memoization in pure functional languages that is described in Field and Harrison [1988].

The conventional notion of memoization

The conventional notion of memoization [Michie 1968, Hughes 1985] involves a process by which a function is made to automatically memorize and subsequently recall all results computed. The conventional implementation of the technique involves the maintenance of a memo-table of previously computed (input, result) pairs for each function to be memoized. When a memoized function is applied, it begins by referring to its memo-table. If the input has been processed before, the previously computed result is returned. If the input has not been processed before, then

1. the result is computed using the original definition of the function with the exception that all recursive calls use the memoized function,
2. the memo-table is updated with the newly computed result, and
3. the result is returned.

Memoization can result in an improvement in efficiency and in some cases an improvement in complexity owing to the fact that a memoized function never recomputes any result.

Norvig's use of memoization to improve the efficiency of top-down parsing

Norvig has shown how memoization can be elegantly implemented in Common Lisp. In Norvig's approach a function `f` can be memoized by applying a function `memoize` to it. The function `memoize` changes the global definition of `f` so that when it is applied, it refers to a table of previously computed (input, result) pairs. This table is part of the definition of the "memoized" version of `f`. Update of the table occurs on every call of `f`. In this approach, both the process of memoizing a function, and the process of updating the memo-table make use of Common Lisp's mutable function name space.

* Support received from NSERC grant number OGP0009181.

Norvig shows how a parser with the same asymptotic behavior as Earley's algorithm can be obtained by applying the function memoize to a simple 15 line top-down fully-backtracking parser generator. Norvig's memoized parser generator cannot accommodate left-recursive productions but is as general as Earley's algorithm in all other respects.

Memoization at the source code level in pure functional programming

For the reasons discussed above, Norvig's approach cannot be implemented directly in a pure functional programming language. However, we can adapt Norvig's approach for such use if we employ an alternative technique for implementing memoization that has been described by Field and Harrison [1988]. This technique is similar to conventional approaches to implementing memoization, but differs in that instead of associating memo-tables with functions, memo-tables are associated with the inputs to and outputs from functions.

According to the technique described by Field and Harrison, a function `f` in a pure functional programming language can be memoized by modifying its definition so that:

1. `f` accepts a memo-table as part of its input and returns a memo-table as part of its output. The memo-table in the output from `f` contains all entries in the input table together with any new (input, result) pairs that were generated during the execution of the body of `f`.
2. Before `f` computes a result, it examines the memo-table in its input to see if the result has already been computed by a previous application.
3. The memo-table that forms part of the output from any recursive call of `f` is used as part of the input to the subsequent recursive call of `f`, if such a call exists.

To illustrate this approach we show how the Fibonacci function can be memoized. We begin with the textbook definition of this function:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

If implemented directly in a pure functional programming language, this function has exponential complexity. The cause of the exponential behavior is the duplication of computation in the two recursive calls of fib. This duplication can be avoided by memoization using the technique described above. The resulting memoized version of the function, which has linear complexity, can be coded in the pure functional programming language Miranda[2] as follows:

```
memo_fib (0, mt)  = (1,(0,1) : mt)
memo_fib (1, mt)  = (1,(1,1) : mt)
memo_fib (n, mt)
     = (memorized_res, mt), if memorized_res  ~= -1
     = (res1 + res2, (n, res1 + res2): updated_table2), if memorized_res = -1
       where
       (res1, updated_table1) = memo_fib (n - 1, mt)
       (res2, updated_table2) = memo_fib (n - 2, updated_table1)
       memorized_res = -1          , if look_up  = []
       memorized_res = (look_up)!0 , if look_up ~= []
       look_up = [r | (x,r) <- mt; x = n]
```

The initial call of `memo_fib` is applied to a pair consisting of the number whose Fibonacci value is required and an empty memo-table, eg

```
memo_fib (4, []) => (5,[(4,5),(3,3),(1,1),(2,2),(0,1),(1,1)])
```

[2] Miranda is a trademark of Research Software Ltd.

This approach to memoization is not as elegant as Norvig's, but has the advantage that it does not compromise referential transparency and can therefore be used in a pure functional programming style. Readers should note that we have used the example above only to illustrate this approach to memoization and that there are other ways to transform the original version of the Fibonacci function to linear form which result in much less convoluted definitions.

Implementing top-down non-deterministic language processors as executable specifications in pure functional programming languages

The conventional approach to implementing modular top-down backtracking parsers in a pure functional programming language is to define three higher-order functions, which we shall refer to as `term`,`orelse` and `then`, which can be used to construct language processors as executable specifications of grammars. This approach was originally developed by Burge [1975] and has been recently extended to handle left-recursive productions by Frost [1993].

To simplify description of the technique we describe it with respect to the construction of recognizers rather than parsers. The modification to generate parse tables is not too difficult.

The "recognizer building" functions `term`, `orelse` and `then` can be coded in Miranda as follows:

```
term c input  = concat (map test_for_c input)
                    where
                    test_for_c (t:ts) = [ts], if t = c
                    test_for_c  any    = []

(x $orelse y) input  = x input ++ y input

(x $then y) inp = [inp2| inp1 <- x inp; inp2 <- y [inp1]]
```

Executable specifications of recognizers built with these functions have a structure that is similar to that of the corresponding BNF grammar. For example:

```
a          = term  'a'
end        = term  '.'
end'       = term  ';'
as         = a
             $orelse
             (a $then as)
asas       = as    $then as
expr1      = asas  $then end
expr2      = asas  $then end'
alt_expr   = expr1 $orelse expr2
```

The following are examples of the use of these recognizers:

```
as ["aaa"]   => ["aa", "a", ""]
asas ["aaa"] => ["a", "", ""]
```

The first example shows that "aaa" can be recognized three ways as an `as`. The first way uses only one of the a's and leaves two a's to be processed. The second way uses all two a's and leaves one. The third way uses all three a's.

The second example shows that "aaa" can be recognized as an `asas`, in three ways.

Recognizers that are built in the above fashion have exponential complexity in the worst case.

Memoizing purely functional top-down non-deterministic language processors.

We now show how Norvig's approach can be adapted, using a variation of the technique described by Field and Harrison, such that purely functional top-down nondeterministic language processors can be memoized to achieve polynomial complexity. The approach is relatively straightforward:

1. All recognizers take a pair as input and a pair as output. The second element of each pair is a memo-table which may or may not be used and/or updated by the recognizer.
2. The functions `term, orelse` and `then` are modified so that recognizers that are built with them pass the memo-table from one application of a recognizer to the next so that no information is lost.
3. A function memoize is defined which takes a recognizer name and a recognizer as input and which returns a memoized recognizer as result.

The implementation of the technique, which is illustrated by the following definitions in Miranda, is significantly more complicated than in Norvig's implementation in Common Lisp because of the necessity of passing the memo-table as argument to and result from recognizers.

We begin with some type declarations:

```
|| ************************************************************************
|| Type declarations

memo_table          == [(num, terminal,
                         [(recognizer_name, [(begin_point,end_point)])])]

terminal            == char
recognizer_name     == [char]
end_point           == num
begin_point         == num
mt_index            == num
input               == ([(begin_point, end_point)], memo_table)
output              == ([(begin_point, end_point)], memo_table)
recognizer          == input -> output
```

Next, we define the functions that implement the memoization:

```
||**********************************************************************
|| memo_apply takes a recognizer as input and returns a recognizer as
|| result. The returned recognizer is the same as the input but differs
|| in that  the memo-table given as part of the input is passed from one
|| application of the recognizer to the next when the recognizer is
|| applied to an input that contains a list of (begin,end) pairs.
|| This ensures that all of the applications of the recognizer are added
|| to the table if the recognizer or any of its components are memoized

memo_apply :: recognizer -> recognizer

memo_apply  rec ([], mt) = ([], mt)
memo_apply  rec ((b,e):rest, mt)
                        = (mkset (r1 ++ rs),mtrs)
                          where
                          (r1, mt1)  = rec ([(b,e)], mt)
                          (rs, mtrs) = memo_apply rec (rest, mt1)
```

```
||*************************************************************************
|| memoize takes a recognizer name and a recognizer as input and returns
|| a memoized recognizer as result. The memoized recognizer looks up
|| results in the memo-table given in its input and updates the table if
|| the result has not been computed before, before returning it as output


memoize :: recognizer_name -> recognizer -> recognizer

memoize rec_name rec ([], mt) = ([],mt)
memoize rec_name rec ((b,e) : inps, mt)
 = (mkset (r1 ++ rs), mtrs)
   where
   (r1, mt1)   = (memo_res!0, mt)                          ,if memo_res ~= []
               =(r1',update_mt (e + 1) rec_name r1' mt1'),if memo_res = []
   (r1', mt1') = rec ([(b,e)], mt)
   (rs, mtrs)  = memoize rec_name rec (inps, mt1)
   memo_res    = lookup (e + 1) rec_name mt
```

The following functions are used for lookup and update of the memo-tables:

```
|| *************************************************************************
|| lookup takes an index, a recognizer name and a memo table as input
|| returns a list of pairs as result, each pair indicates the location
|| of a beginning and an end point of a successful application of the
|| recognizer to the input

lookup :: mt_index -> recognizer_name -> memo_table
          -> [[(begin_point, end_point)]]

lookup index rec_name mt
   = [pairs | (n, pairs) <- third (mt!index); n = rec_name]


|| *************************************************************************
|| The function update updates a memo-table to create a new memo-table

update_mt :: mt_index -> recognizer_name -> [(begin_point, end_point)]
             -> memo_table -> memo_table

update_mt index  rec_name pairs mt
        = map (update_mt_entry index rec_name pairs) mt

third (a,b,c) = c

update_mt_entry index rec_name pairs (n, term, list)
  = (n, term, add_rec rec_name pairs list), if n = index
  = (n, term, list)                       , otherwise

add_rec rec_name pairs list_of_recs
  =  (rec_name, pairs) : list_of_recs
```

We now redefine `term`, `orelse` and `then`:

```
||*************************************************************************
|| term takes a terminal as input and returns a recognizer for that
|| terminal as result. It differs from the standard definition
|| only in that it takes a memo-table as part of its input

term :: terminal -> recognizer

term c (inp, mt)
 = ([(e + 1,e + 1) | (b,e) <- inp; second (mt!(e+1)) = c], mt)

second (x, y, z) = y

||*************************************************************************
|| the function memo_orelse is defined as an infix operator which
|| takes two recognizers as input and returns a reconizer as result

memo_orelse ::  recognizer -> recognizer -> recognizer

(p $memo_orelse  q) (inp_pairs, mt)
       = (mkset (rp ++ rq), mtq)
         where
         (rp, mtp) = memo_apply p (inp_pairs, mt)
         (rq, mtq) = memo_apply q (inp_pairs, mtp)
||*************************************************************************
|| the function memo_then is defined as an infix operator which
|| takes two recognizers as input and returns a reconizer as result

memo_then ::  recognizer -> recognizer -> recognizer

(p $memo_then q) ([], mt) = ([], mt)
memo_then p q ((b,e):inps,mt)
  = (mkset (r1 ++ rs), mtrs)
    where
    (r1, mt1)  = (p $one_then q) ([(b,e)], mt)
    (rs, mtrs) = memo_then  p q (inps, mt1)
    (p $one_then q) ([(b,e)], mt)
       = (mkset(map (change_begin_points (e + 1)) rq), mtq)
          where
          (rp, mtp)  = p ([(b,e)], mt)
          (rq, mtq)  = memo_apply q (rp, mtp)
          change_begin_points e (x,y) = (e, y)
```

The following are memoized versions of the recognizers presnted in the previous section:

```
||*************************************************************************

a          = term 'a'
end        = term '.'
end'       = term ';'
as         = memoize "as"  (a
                            $memo_orelse
                            (a $memo_then  as))
asas       = memoize "asas"  (as   $memo_then  as)
expr1      = memoize "expr1" (asas $memo_then  end)
expr2      = memoize "expr2" (asas $memo_then  end')
alt_expr   = memoize "alt_expr"  (expr1
                                  $memo_orelse
                                  expr2)
```

The following illustrates use of the memoized recognizers:

```
||*************************************************************************
|| aaply_rec takes a recognizer and a character string as input,
|| expands the string and then applies the recognizer to it

apply_rec rec inp
  = rec ([(-1,-1)], expanded_inp)
    where
    expanded_inp = [(n,t,[]) | (n,t )<- (zip2 [0..] (inp ++ "@"))]

||*************************************************************************
|| In each of the following example applications, the output is a pair
|| consisting of a list of (begin,end) points indicating a successful
|| application of the recogniser, followed by the final memo-table
|| The last entry in each memo-table is for the terminating terminal '@'
|| which is added to each input to simplify the algorithms


test1   = apply_rec a         "abc"

|| test1 => ([(0,0)],
||               [(0,'a',[]),
||                (1,'b',[]),
||                (2,'c',[]),
||                (3,'@',[])])

test2   = apply_rec as        "aaa."

|| test2 => ([(0,0),(0,1),(0,2)],
||               [(0,'a',[("as",[(0,0),(0,1),(0,2)])]),
||                (1,'a',[("as",[(1,1),(1,2)])]),
||                (2,'a',[("as",[(2,2)])]),
||                (3,'.',[("as",[])]),
||                (4,'@',[])])

test3   = apply_rec alt_expr  "aaa;"

|| test3 => ([(0,3)],
||              [(0,'a',[("alt_expr",[(0,3)]),
||                       ("expr2",[(0,3)]),
||                       ("expr1",[]),
||                       ("asas",[(0,1),(0,2)]),
||                       ("as",[(0,0),(0,1),(0,2)])]),
||               (1,'a',[("as",[(1,1),(1,2)])]),
||               (2,'a',[("as",[(2,2)])]),
||               (3,';',[("as",[]),("as",[])]),
||               (4,'@',[])])
```

Concluding comments

Using the technique described in this note, it is possible to build top-down non-deterministic language processors as modular executable specifications in pure functional programming languages, ie languages that admit no mutable objects. These language processors have polynomial complexity in the worst case. It is an open question as to whether this technique can be integrated with the technique for accommodating left-recursive productions in top-down parsing as described by Frost [1993]. If it can be, then the technique can be used to build efficient language processors as modular executable specifications of arbitrary context-free grammars.

A number of advantages derive from this approach to building language processors:

1. Language processors constructed in this way are extremely modular.
2. The major disadvantage of top-down non-deterministic parsing with full backtracking, ie exponential behavior, is overcome.
3. The memoization function can be applied selectively to components of language processors not just to those parts that correspond to non-terminals in the grammar. For example, suppose we have the following grammar:

```
x ::= a then b then c
y ::= a then b then d
```

   In order to fully eliminate recomputation, it would be appropriate to memoize the recognizer corresponding to `a then b` in addition to those for `x` and `y`. However, it would be inappropriate to memoize recognizers of terminals because the memo-table operations would be more costly than simply examining the terminal.

The technique described in this paper is currently being incorporated into the Windsor attribute grammar programming environment W/AGE [Frost and Karamatos 1993].

References

Burge, W. H. (1975) *Recursive Programming Techniques*, Addison-Wesley Publishing Company, Reading, Massachusetts.

Field, A. J. and Harrison, P. G. (1988) *Functional Programming*, Addison-Wesley Publishing Company, Reading, Massachusetts.

Frost, R. A. (1993) Guarded attribute grammars, *Software Practice & Experience*, 23 (10) 1139 - 1156.

Frost, R. A. and Karamatos, S. (1993) Supporting the attribute grammar programming paradigm in a lazy functional programing language, *International Lecture Series on Functional Programming, Concurrency, Simulation and Automated Reasoning*. McMaster University 1991. Springer-Verlag Lecture Note Series 693, editor P. Lauer. 278 - 295.

Hughes, R. J. M. (1985) Lazy memo functions. In proceedings, *Conference on Functional Programming and Computer Architecture* Nancy, France, September 1985. Springer-Verlag Lecture Note Series 201, editors G. Goos and J. Hartmanis, 129 - 146.

Michie, D. (1968) 'Memo' functions and machine learning, *Nature*, 218, 19 - 22.

Norvig, P. (1991) Techniques for automatic memoisation with applications to context-free parsing, *Computational Linguistics*, 17 (1), 91 - 98.