Monadic Memoization towards Correctness-Preserving Reduction of Search

Richard Frost

School of Computer Science, University of Windsor
Ontario, Canada
richard@uwindsor.ca

Abstract. Memoization is a well-known method which makes use of a table of previously-computed results in order to ensure that parts of a search (or computation) space are not revisited. A new technique is presented which enables the systematic and selective memoization of a wide range of algorithms. The technique overcomes disadvantages of previous approaches. In particular, the proposed technique can help programmers avoid mistakes that can result in sub-optimal use of memoization. In addition, the resulting memoized programs are amenable to analysis using equational reasoning. It is anticipated that further work will lead to proof of correctness of the proposed memoization technique.

1 Introduction

Search is ubiquitous in artificial intelligence. For many difficult problems, search time grows exponentially with respect to the problem size. In many cases, the time can be significantly reduced by making sure that parts of the search space are not revisited unnecessarily. In some cases a reduction in complexity is possible.

1.1 The Need for Selective Memoization

Some parts of a program can benefit from memoization whereas other parts will not. For example, the time complexity of the following naive Fibonacci program can be reduced from exponential to liner through memoization:

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

The reason for the improvement is that in the unmemoized form, the second call of `fib (n - 2)` repeats a computation carried out by the first call `fib (n - 1)`. Expansion of the computation tree will illustrate the extent of recomputation that can be prevented by memoization. On the other hand, consider a program that simply returns the first element of a list. This operation has constant complexity. Memoization would require that the list input be compared with lists

Y. Xiang and B. Chaib-draa (Eds.): AI 2003, LNAI 2671, pp. 66–80, 2003.

used as keys in the memo table. The complexity would now be O(length of the list).

A solution, therefore, is to provide the programmer with a function *memoize* which can be used to memoize selected parts of the program. The function *memoize* takes care of updating and using the memo tables when the memoized parts of the program are executed.

1.2 The Need for Pure Functionality?

One should only memoize parts of the program that are purely functional in the sense that the result should only depend on the input arguments, and there should be no side-effects, The reason for this is that memo-table lookup uses inputs as keys. If the result depends on other values that are accessible through non-functional calls, then the memo-table will return the wrong result. Also, if a component has side effects, such as having a subcomponent that updates a counter, then memoization will corrupt those side effects.

Therefore, a necessary, but not sufficient requirement for correct memoization, is that the components to be memoized must be purely functional. We use the term "correct" in the sense that a correct memoization process should not change the results returned by a program, or its termination properties. Many search algorithms are very complicated and programmers will be tempted to use non-functional features in their implementation. For example, the use of updateable global variables for various types of bookkeeping. It is difficult to determine which parts of a complicated program are purely functional, and can therefore be safely memoized. One solution to this is to use a pure-functional programming language such as Miranda (Turner 1985) or Haskell (Hudak et.al. 1992) to implement complicated search algorithms.

1.3 Memoization in a Pure-Functional Language

Use of a pure-functional programming language prevents certain types of inaccurrate use of memoization, and it also facilitates the construction of a memoization process that is amenable to analysis through equational reasoning, but it has a major disadvantage: memo tables cannot be implemented as updateable stores. Update is a side effect and is not allowed in a pure-functional language.

The solution is to provide the memo table as an extra argument to functions that are to be memoized. Memoized functions now begin by checking to see if the original input argument has an entry in the memo-table given as the additional input argument. If it has, then that result is returned with the original memo-table. If not, the body of the function is executed and the result is returned together with a new memo-table containing the additional entry. This is not as inefficient as it might first appear. Pure-functional programming languages use pointers to re-use parts of input arguments; this is safe owing to the fact that there is no update and therefore no possibility of corruption of a value that is pointed to.

1.4 The Need for a more Formal Technique

Two problems remain. Firstly, it is possible to make a mistake in threading the memo table through components resulting in sub-optimal use of memoization. For example, consider the following, which uses notation that is defined more fully in section 2.

```
(p $then q) inp =  []     , if rp  = []
                =  q rp   , otherwise
                   where  rp = p inp
```

This defines an infix higher-order operator `then` which takes two functions `p` and `q` as input and which returns a function `h` as result, such that when that function is applied to some input `inp` it begins by computing the value `rp` by applying `p` to `inp`. If `rp` is an empty list, then `h` returns an empty list as result. If `r` is not empty, then `h` returns the result obtained by applying `q` to `rp`.

The higher-order function `then` is similar to the operator of function composition, but differs in that the second function is not applied if the first function returns an empty list as result. This kind of operator can be used to prevent infinite looping that would otherwise occur when certain mutually-recursively defined functions are composed. Revising the definition of `then` in order to thread memo tables through `p` and `q` results in the following:

```
(p $mthen q) (inp, table) =  ([], table), if rp = []
                          =  q (rp, tp), otherwise
                             where
                             (rp, tp) = p (inp, table)
```

Even with this relatively simple definition, we have made a mistake. If the function `p` returns an empty list, possibly signifying that it failed in some sense when applied to the input, it may have done some work that has been recorded in the memo table `tp` returned as result. The definition above loses the results of this work. The correction is to replace `([], table)` with `([], tp)`. We show later that this error can result in exponential behavior for a top-down fully-backtracking language recognizer that would otherwise have cubic complexity after memoization.

A second problem is that the ad hoc memoization process can result in other errors which corrupt the result returned. The memoization process described so far requires that every function which is to be memoized, or which interacts with a function to be memoized, must be modified to accept the extra memo table input and return the table as part of its output.

To overcome these shortcomings, a more structured/formal technique for memoizing the search algorithm is required. We need to guarantee that the memo-tables are threaded through appropriate components of the search algorithm, and we need to prove that the memoization process preserves the correctness of the search algorithm.

1.5 Monadic Memoization

According to Wadler (1990), monads were introduced to computing science by Moggi (1989) who noticed that reasoning about programs that involve handling of state, exceptions, I/O, or non-determinism can be simplified, if these features are expressed using monads. Inspired by Moggi's ideas, Wadler proposed monads as a way of systematically adding such features to algorithms. The main idea behind monads is to distinguish between the type of values and the type of computations that deliver these values. This paper shows how monads can be used to systematically memoize search algorithms in a way that facilitates proof of correctness of the memoization process. A monad is a triple (`M, unit, bind`) where `M` is a type constructor, and *unit* `and` *bind* are two polymorphic functions. `M` can be thought of as a function on types, that maps the type of values into the type of computations producing these values. `unit` is a function that takes a value and returns a corresponding computation; the type of `unit` is `a -> Ma`. The function `bind` represents sequencing of two computations where the value returned by the first computation is made available to the second (and possibly subsequent) computation. The type of `bind` is

```
Ma -> (a -> Mb) -> Mb
```

In order to use monads to provide a structured method for adding new effects to a functional program, we begin by identifying all functions that will be involved in those effects. We then replace those functions, which can be of any type `a -> b`, by functions of type `a -> Mb`. In effect, we change the program so that selected function applications return a computation on a value rather than the value itself. This computation may be used to add features such as state to the program. In order to effect this change, we use the function `unit` to convert values into computations that return the value but do not contribute to the new effects, and the function `bind` is used to apply a function of type `a ->Mb` to a computation of type `Ma`. Having made these changes, the original program can be obtained by using the identity monad `idm`, as defined later. In order to memoize the program we simply replace the identity monad with the state monad `stm` and make a few local changes as required to the rest of the program.

1.6 Advantages of this Approach

The technique facilitates the systematic and controlled use of memoization in search algorithms, such that:

1. The approach is purely functional and equational reasoning can be used to analyze the program and to prove correctness of the memoization process. The use of monads further facilitates such proof.
2. As with many other proposed methods for memoization, the use of a "memoize" function allows the search engineer to selectively memoize parts of the search algorithm.
3. The approach helps to avoid mistakes in threading memo-tables through functions in the correct order and therefore facilitates optimal use of memoization.

1.7 Structure of the Rest of the Paper

We begin by briefly describing related work. We then introduce some notation. We follow this with a description of how monadic memoization can be used to improve the complexity of the naive Fibonacci program. At first, the reader may think that we are developing some heavy-duty techniques to achieve what can be achieved by a simple rewrite of the naive Fibonacci program. However, section 8 shows that **the same monad, and the same memoization function** can be used to systematically and easily improve the complexity of relatively complicated definitions of operators that can be used to quickly build top-down fully-backtracking language recognizers. Such recognizers are of interest in their own right as they are highly modular, simple to construct, and can accommodate ambiguous grammars. We then sketch out a proof of correctness of the memoization process. We conclude by discussing how the approach can be applied to other search problems.

2 Overview of Related Work

1. Memoization has a long history. Michie (1968) and Hughes (1985) are two of many publications on this subject.
2. Threading of memo tables though functional programs has been described by Field and Harrison (1988) and investigated in detail by Khoshnevisan (1990).
3. Wadler (1990) introduced the use of monads to add effects to pure functional programs. Subsequent publications (e.g. Wadler,1995) include discussion of the use of monads to structure functional recognizers and evaluators.
4. Koskimies (1990) convincingly explains how how top-down fully-backtracking language processors are considerably more modular than those constructed with alternative search strategies.
5. Norvig (1991) shows how memoization of top-down fully-backtracking language recognizers written in LISP results in processors that are as efficient and as general as Earley's algorithm, with the exception that they cannot accommodate. left-recursive productions. Norvig's recognizers have cubic complexity compared to exponential behavior of the unmemoized versions. Selective memoization is achieved through use of a memoization function defined in terms of an updateable memo-table. Owing to the fact that memoization involves updateable state Norvig's approach requires considerably more complex apparatus than equational reasoning to prove correctness of the memoization process.
6. The use of higher-order functions (combinators) to build functional language processors, which is described in detail in section 7, was originally proposed by Burge (1975) and further developed by Wadler (1985) and Fairburn (1986). It was first used to build evaluators for ambiguous natural-language by Frost and Launchbury (1989). It is now frequently used by the functional-programming community for language prototyping and natural-language processing. In the following, we describe the approach with respect

to language recognizers although the technique can be readily extended to parsers, syntax-directed evaluators and executable specifications of attribute grammars (Frost 1992 and 2002, Augusteijn 1993, and Leermakers 1993).

7. Leermakers (1993), Frost (1993), and Johnson (1995) have described different techniques by which the functional approach to building top-down backtracking language processors can be extended to accommodate left-recursive productions. It is interesting to note that Johnson's approach uses memoization, together with continuation-passing-style programming, to achieve efficiency and accommodate left-recursion.
8. Frost and Szydlowski (1995) show how purely-functional top-down backtracking language processors can be memoized, and proved that time complexity can be reduced from exponential to cubic.
9. The use of monads to systematically memoize purely-functional top-down recognizers was suggested to the author of this paper by an anonymous reviewer of the paper by Frost and Szydlowski. The reviewer identified the mistake in the threading of memo-tables through the `$then` operator as discussed earlier, and pointed out that this would result in exponential behaviour for certain inputs. A brief discussion of the potential use of monads in memoization of pure-functional recognizers was given at the end of their revised paper.
10. Panitz (1996) has developed a technique for proving termination for lazy functional languages by abstract reduction, and has used this technique to prove termination for a sub-set of recognizers that can be constructed using the combinators of Frost and Launchbury, a variation of which are used in section 7.

3 Notation

We use the notation of the programming language Miranda[1] (Turner 1985), rather than a functional pseudo-code, in order that readers can experiment with the definitions directly. The technique can be implemented in other languages.

- `f = e` defines `f` to be a constant-valued function equal to the expression `e`.
- `f a1 ... an = e` can be loosely read as defining `f` to be a function of `n` arguments whose value is the expression `e`. However, Miranda is a fully higher-order language — functions can be passed as parameters and returned as results. Every function of two or more arguments is actually a higher order function, and the correct reading of `f a1 ... an = e` is that it defines `f` to be a higher-order function, which when *partially-applied* to input `i` returns a function `f' a2 ... an = e'`, where `e'` is `e` with the substitution of `i` for `a1`.
- The notation for function application is simply juxtaposition, as in `f x`. Function application has higher precedence than any operator.

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

- Function application is left associative. For example, `f x y` is parsed as `(f x) y`, meaning that the result of applying `f` to `x` is a function which is then applied to `y`. Round brackets are used to override the left-associative order of function application. For example, the evaluation of `f (x y)` requires `x` to be applied to `y`, and then `f` to be applied to the result.
- In a function definition, the applicable equation is chosen through pattern matching on the left-hand side in order from top to bottom, together with the use of guards following the keyword `if`.
- Round brackets with commas are used to create tuples, e.g. `(x, y)` is a binary tuple. Square brackets and commas are used to create lists, e.g. `[x, y, z]`. The empty list is denoted by `[]` and the notation `x : y` denotes the list obtained by adding the element `x` to the front of the list `y`. The notation `"x1 .. xn"` is shorthand for `['x1', .. ,'xn']`
- `t1 -> t2` is the type of functions with input type `t1` and output type `t2`. `f :: e` states that `f` is of type `e, and t1 == t2` declares t1 and t2 to be type synonyms.
- The notation `x => y` means that `y` is the result of evaluating `x`.

4 Rewriting the Fibonacci Program in Monadic Form – The Identity Monad

We begin by defining the identity monad `idm` (Wadler 1995), in which the computation simply returns its value, and `bind` is `postfix` function application. The star `*` means any type.

```
idm * == *

unit1 :: * -> idm *
unit1 x = x

bind1 :: idm * -> (* -> idm **) -> idm **
(p $bind1 k) = k p
```

We can use this monad to restructure the naive Fibonacci program given in section 1. This is the first step towards memoization:

```
fib1 0 = unit1 1
fib1 1 = unit1 1
fib1 n = fib1 (n - 1) $bind1 f
         where f a =  fib1 (n - 2) $bind1 g
                      where g b = unit1 (a + b)
```

Such restructuring is relatively "clerical": Firstly, we apply `unit1` to all values which are returned by expressions that do not include calls to the `fib` function. Secondly, we analyze the expression `fib (n - 1) + fib(n - 2)` and work out an order of computation: begin with `fib (n - 1)` bind this result into the next

computation `f` which involves `fib (n - 2)`, bind the result into the next part of the program `g` which returns a computation obtained by applying `unit1` to the result of the addition. The resulting program `fib1` acts like `fib` in all respects. Simple equational rewriting shows that `fib` and `fib1` are equal.

5 Adding a Counter to the Fibonacci Program – The State Monad

Before we memoize the `fib` function, we show how to add a counter to it using the state monad `stm` (Wadler 1995) defined as follows:

```
stm * == state -> (*, state)
state == num
unit2 :: * -> stm *
unit2 a = f where f t = (a, t)
bind2 :: stm * -> (* -> stm **) -> stm **
(m $bind2 k) = f
               where f x = (b, z)
                           where (b, z) = k a y
                                          where (a, y) = m x
```

In this case, the state is a numeric counter. The function `unit2` takes a value `v` and returns a computation of type `state -> (*, state)`, which takes a state as input and returns the value `v` paired with the state unchanged. For example when `unit2 5` is applied to state `6` (the counter), the result returned is `(5, 6)`.

The operator (infix function) `$bind2` is a little difficult to understand at first. Roughly, it takes a computation `m` which "involves" a value `v` of type `*` as one operand, and a function `k` of type `(* -> stm **)` as the other operand. It picks out the value `v`, applies `k` to it, and returns a computation that when applied to state returns a pair consisting of the result of the function application and the state unchanged. Basically, `$bind` creates a computation that threads the state through its components. To add a counter to the `fib` function, we simply replace the identity monad with the state monad, and make a small change so that the counter is incremented each time `cfib` is called:

```
fib 0 = unit2 1
fib 1 = unit2 1
fib n = cfib (n - 1) $bind2 f
        where f a =  cfib (n - 2) $bind2 g
                     where g b = unit2 (a + b)

cfib   =  count fib

count f n c =  (res, k + 1)
               where
               (res, k) = f n c
```

The function `cfib` computes the same value as before paired with the count. For example:

```
cfib 20 0 => (10946,21891)
```

6 Memoizing the Fibonacci Program

In order to memoize the `fib` function, we use the same state monad `stm`, change the type of the state from a numeric counter to a memo-table of type `[([char], [(num, num)])]`, and replace the count function with the function `memoize`. Note that the definition of `unit2` and `bind2` remain the same.

```
state == [([char],[(num, num)])]
fib 0 = unit2 1
fib 1 = unit2 1
fib n = mfib (n - 1) $bind2 f
        where f a =  mfib (n - 2) $bind2 g
                     where g b = unit2 (a + b)

mfib   =  memoize "fib" fib
```

The memoized function `mfib` has liner time and space complexity, compared to exponential behavior of the original `fib` program. To illustrate the flexibility of this approach. The definition above can be modified so that only the left-branching call of `fib` is memoized. Interestingly, the execution data for the resulting program `left_mfib` shown on the next page, suggests that such memoization results in polynomial complexity.

```
fib 20 => 10946                                          time  = 235015
fib 21 => 17711                                          time  = 380274
fib 22 => 28657                                          time  = 615308
fib 50 => ran out of heap space

mfib 20  [] => (10946, updated table)                    time  = 5494
mfib 21  [] => (17711, updated table)                    time  = 5775
mfib 22  [] => (28657, updated table)                    time  = 6056
mfib 200 [] => (453973694165307953197296969697410619233826,
                              updated table)             time  = 41046

left_mfib 20 []  => (33429, updated table)               time = 19171
left_mfib 50 []  => (20365011074, updated table)         time = 165170
left_mfib 100 [] => (573147844013817084101, table)       time = 1034095
left_mfib 200 [] => (453973694165307953197296969697410619233826,
                              updated table)             time  = 7133195
```

The memo table in this case is of type `[([char],[(num, num)])]`, i.e. a list of pairs. Each pair contains a string of characters identifying a function that has been memoized, followed by a list containing pairs of input/output values computed for that function. The reason for having multiple pairs in the table is that this allows a number of different functions in one program to be memoized and their results stored in a single table. We make use of this feature in later examples. The `memoize` function, defined below, creates a new function from the function to be memoized, such that the new function performs lookup and update operations on the memo table being threaded through the computation.

```
memoize name f inp table
   = (res, update newtable name inp res), if (table_res = [])
   = (table_res!0, table)                , otherwise
     where table_res       = lookup name inp table
           (res, newtable) = f inp table

lookup name inp table
  = [], if res_in_table = []
  = [res | (i, res) <- (res_in_table!0); i = inp], otherwise
    where
    res_in_table = [pairs | (n, pairs) <- table; n = name]

update [] name inp res  = [(name,[(inp, res)])]
update ((key, pairs):rest) name inp res
  = (key,(inp,res):pairs):rest, if key = name
  = (key,pairs): update rest name inp res, otherwise
```

Linear space complexity can be achieved by a simple modification to the memoize function to keep only the two most-recently computed values in the memo table.

7 A Modular Top-Down Fully-Backtracking Language Recognizer

One approach to implementing language processors in a modern functional programming language is to define a number of higher-order functions (combinators) which, when used as infix operators (denoted in this paper by the prefix `$`), enable processors to be built with structures that have a direct correspondence to the grammars defining the languages to be processed. For example, the function `s`, defined below is a recognizer for the language defined by the grammar `s ::= 'a' s s | empty`

```
s =  (a $then s $then s) $orelse empty
a =  term  'a'
```

The combinators that we use in this paper are from Frost and Launchbury (1989):

```
recognizer  == [char] -> [[char]]

term   :: char -> recognizer
term c   []    = []
term c (t:ts)  = [ts], if t  = c
               = []  , otherwise

orelse :: recognizer -> recognizer -> recognizer
(p $orelse q) inp =   unite (p inp) (q inp)

then   :: recognizer -> recognizer -> recognizer
(p $then q)  inp =  apply_to_all q (p inp)

apply_to_all q []     = []
apply_to_all q (r:rs) =  unite (q r) (apply_to_all q rs)

empty x = [x]

unite x y = mkset (x ++ y)
```

According to the approach, a recognizer is a function mapping an input string to a list of outputs. The input is a sequence of tokens to be analyzed. Each entry in the output list is a sequence of tokens yet to be processed. Using the notion of "failure as a list of successes" (Wadler 1985) an empty output list signifies that a recognizer has failed to recognize the input. Multiple entries in the output occur when the input is ambiguous. In the examples in this paper it is assumed that all tokens are single characters.

The simplest type of recognizer is one that recognizes a single token at the beginning of a sequence of tokens. Such recognizers may be constructed using the higher-order function `term` defined above. The following illustrates use of `term` in the construction of a recognizer for the character `'c'`. The empty list in the second example signifies that `c` failed to recognize a token `'c'` at the beginning of the input

```
c        =  term 'c'
c "cxyz" => ["xyz"]
c "xyz"  => []
```

Alternate recognizers may be built using the higher-order function `orelse` as defined above. When a recognizer `p $orelse q` is applied to an input `inp`, the value returned is computed by uniting the results returned by the separate application of `p` to `inp` and `q` to `inp`. The following illustrates use of `orelse` in the construction of a recognizer `c_or_d` and the subsequent application of this recognizer to three inputs.

```
c_or_d  = c $orelse d
c_or_d "cxyz" => ["xyz"]
c_or_d "abc"  => []
```

Sequencing of recognizers is obtained through use of the higher-order function `then` defined as above. When a recognizer `p $then q` is applied to an input `inp`, the result returned is a list obtained by applying `q` to each of the results in the list returned by `p`. The following illustrates use of `then` in the construction of a recognizer `c_then_d`, and the subsequent application of `c_then_d` to two inputs:

```
c_then_d = c $then d
c_then_d "cdxy" => ["xy"]
c_then_d "cxyz" => []
```

The "empty" recognizer always succeeds and returns a singelton list containing the input. The `unite` operation removes duplicates in the results returned by `orelse` and `then`. The example application given below illustrates use of the recognizer `s` from the previous page, and shows that the prefixes of the input ‘‘`aaa`’’ can be successfully recognized in different ways. The empty string in the output, denoted by `""`, corresponds to cases where the whole input ‘‘`aaa`’’ has been recognized as an `s`.

```
s   "aaa"  => ["","a","aa","aaa"]
```

Recognizers constructed in this way are easy to construct and are highly modular, but have exponential time complexity.

8 Use of the State Monad to Memoize Language Recognizers

The advantage of the proposed approach will now become apparent. In order to memoize recognizers that are constructed using the method described above, we simply rewrite the definitions of the combinators to use the state monad, change the state (memotable) to be of type `[([char],[([char], [[char]])])]` to be compatible with the input/output types of recognizers, and apply the `memoize` function from the Fibonacci example.

```
stm * == state -> (*, state)

state == [([char],[([char], [[char]])])]
term2 c   []    = unit2 []
term2 c (t:ts)  = unit2 [ts], if t  = c
                = unit2 []  , otherwise
(p $orelse2 q) input
               = p input $bind2 f
                 where
                 f a =  q input $bind2 g
                        where
                        g b = unit2 (unite a b)
(p $then2 q)  input = p input }bind2 f
```

```
                      where
                      f a = apply_to_all2 q a
empty2 x = unit2 [x]
apply_to_all2 q []      = unit2 []
apply_to_all2 q (r:rs)  = q r $bind2 f
                          where
                          f a = apply_to_all2 q rs $bind2 h
                                where
                                h b = unit2 (unite a b)


ms = memoize "ms" (a2 $then2 ms then2 ms) $orelse2 empty2
a  =  term2  'a'
```

The memoized recognizer `ms` has worst-case cubic complexity (theoretically this is as good as is possible) compared to the exponential complexity of the original recognizer `s`:

```
s "aaaa" => ["","a","aa","aaa","aaaa"]         time =    5879
s "aaaaaaaaaa"
    => ["","a","aa","aaa","aaaa","aaaaa","aaaaaa","aaaaaaa",
        "aaaaaaaa","aaaaaaaaa","aaaaaaaaaa"] time = 1938151
s "aaaaaaaaaaaaaaaaaaaa"     => ran out of space

ms "aaaaaaaaaa" []           => as above       time =   43642
ms "aaaaaaaaaaaaaaaaaaaa" [] => correct result time =  388837
```

Notice that the structure of the definition of `ms` is the same as the original except for application of the memoize function. The monad and the memo-table function definitions are hidden from the programmer who is constructing the recognizer. Notice also that the programmer can choose which parts of the recognizer to memoize.

9 A Sketch of Proof of Preservation of Correctness

One of the advantages of the proposed approach is that the resulting memoized programs are completely functional and, therefore, equational reasoning can be used in their analysis. In particular, equational reasoning can be used to prove that the memoization process is correct in the sense that termination properties are preserved and that the memoized program returns the same results as the original:

1. Preservation of termination properties can be proven by:
   (a) Showing that the table size is bounded, that update and lookup terminate, and therefore that the `memoize` function terminates for finite input.

(b) Showing that rewriting the program in monadic form does not affect termination properties. For the recognition example, the technique of abstract reduction, which Panitz (1996) developed and has already used to prove termination of a sub-set of recognizers that can be built with the combinators given in the paper, can be used to show that the monadic form of the combinators have the same termination properties.

2. To prove that memoization does not change the results returned, we need to show that the values computed in the monadic form of the program are the same as those in the original, and that update and lookup do not corrupt those values. Although not trivial, it is anticipated that equational reasoning can be readily used to do this.

The resulting proofs apply to any program that is memoized using the technique described in this paper.

10 Concluding Comments

The approach described in this paper can be applied to other types of problem where memoization can be used to avoid reexamining already-visited parts of the search space. Such problems include scheduling, planning, sub-sequence analysis, pattern recognition, database query optimization, theorem proving, computational physics, conformational search in crystallography, and many others. Current work includes construction of the complete proof of preservation of correctness, and investigation of the use of the technique in other search problems.

References

[1] Augusteijn, L. (1993) *Functional Programming, Program Transformations and Compiler Construction*. Philips Research Laboratories. ISBN 90–74445–04–7.

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

[3] Fairburn, J. (1986) Making form follow function: An exercise in functional programming style. *University of Cambridge Computer Laboratory Technical Report* No 89.

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

[5] Frost, R. A. (2002) W/AGE The Windsor Attribute Grammar Programming Environment. *IEEE Symposia on Human Centric Computing Languages and Environments HCC'2002* 96–99.

[6] Frost, R. A. and Szydlowski, B. (1995) Memoizing purely-functional top-down backtracking language processors. *Science of Computer Programming" (27) 263 – 288.*

[7] Frost, R. A. (1993) 'Guarded attribute grammars'.*Software Practice and Experience.*23 (10) 1139–1156.

[8] Frost, R. A. (1992) Constructing programs as executable attribute grammars. *The Computer Journal* 35 (4) 376 – 389.

[9] Frost, R. A. and Launchbury, E. J. (1989) Constructing natural language interpreters in a lazy functional language'. *The Computer Journal – Special edition on Lazy Functional Programming*, **32** (2) 108 – 121.

[10] Hudak, P., Wadler, P., Arvind, Boutel, B., Fairbairn, J., Fasel, J., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Peyton Jones, S., Reeve, M., Wise, D. and Young, J. (1992) Report on the programming language Haskell, a non-strict, purely functional language, Version 1.2 *ACM SIGPLAN Notices* 27 (5).

[11] 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.

[12] Johnson, M. (1995) Squibs and Discussions: Memoization in top-down parsing. *Computational Linguistics* 21 (3) 405–417.

[13] Khoshnevisan, H. (1990) Efficient memo-table management strategies. *Acta Informatica* 28, 43–81.

[14] Koskimies, K. Lazy recursive descent parsing for modular language implementation. *Software Practice and Experience*, 20 (8) 749–772, 1990.

[15] Leermakers, R. (1993) *The Functional Treatment of Parsing*. Kluwer Academic Publishers, ISBN 0–7923–9376–7.

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

[17] Moggi, E. (1989) Computational lambda-calculus and monads. *IEEE Symposium on Logic in Computer Science,* Asilomar, California, June 1989, 14–23.

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

[19] Panitz (1996) Termination proofs for a lazy functional language by abstract interpretation. `citeseer.nj.nec.com/panitz96termination.html`

[20] Turner, D. (1985) A lazy functional programming language with polymorphic types. *Proc. IFIP Int. Conf. on Functional Programmiong Languages and Computer Architecture*. Nancy, France. Springer Verlag Lecture Notes in Computer Science 201.

[21] Wadler, P. (1985) How to replace failure by a list of successes, in P. Jouannaud (ed.) *Functional Programming Languages and Computer Architectures* Lecture Notes in Computer Science 201, Springer-Verlag, Heidelberg, 113.

[22] Wadler, P. (1990) Comprehending monads. *ACM SIGPLAN/SIGACT/SIGART Symposium on Lisp and Functional Programming*, Nice, France, June 1990, 61–78.

[23] Wadler, P. (1995) Monads for functional programming, Proceedings of the Bastad Spring School on Advanced Functional Programming, ed J. Jeuring and E. Meijer. *Springer Verlag Lecture Notes in Computer Science* 925.