Tabled Execution in Scheme
(Scheme Pearl) *

Jeremiah Willcock
Lawrence Livermore National Laboratory
willcock2@llnl.gov

Andrew Lumsdaine
Indiana University
lums@osl.iu.edu

Daniel Quinlan
Lawrence Livermore National Laboratory
dquinlan@llnl.gov

Abstract

Tabled execution is a generalization of memoization developed by the logic programming community. It not only saves results from tabled predicates, but also stores the set of currently active calls to them; tabled execution can thus provide meaningful semantics for programs that seemingly contain infinite recursions with the same arguments. In logic programming, tabled execution is used for many purposes, both for improving the efficiency of programs, and making tasks simpler and more direct to express than with normal logic programs. However, tabled execution is only infrequently applied in mainstream functional languages such as Scheme. We demonstrate an elegant implementation of tabled execution in Scheme, using a mix of continuation-passing style and mutable data. We also show the use of tabled execution in Scheme for a problem in formal language and automata theory, demonstrating that tabled execution can be a valuable tool for Scheme users.

1. Introduction

Tabled execution is a technique introduced in the logic programming community for memoizing the results of predicates, as well as allowing them to be suspended and resumed to prevent infinite recursion (Warren 1992). One advantage of adding tabled execution to a Prolog system is that Prolog plus tabling is a decision procedure for Datalog (Prolog without function symbols), while normal Prolog evaluation causes some Datalog programs to fail to terminate. Other names for tabled execution and similar constructs include OLDT resolution (Tamaki and Sato 1986), SLG resolution (Chen and Warren 1996), and extension tables (Fan and Dietrich 1992). Tabled execution has been used for model checking (Ramakrishna et al. 1997), program analysis (Dawson et al. 1996; Saha and Ramakrishnan 2005), parsing (Warren 1993, 1999), deductive databases (Fan and Dietrich 1992), single-source and all-pairs shortest path algorithms on graphs (Dietrich 1992), and many other problems. The draft book by Warren (1999) provides a good introduction to tabled execution in Prolog, its implementation, and some of its uses. Because of its many uses, it would be beneficial to have a general framework for tabled execution in other languages such as Scheme. The closest available is in (Johnson 1995), which explains how to use code expressing what is effectively tabled execution in Scheme for parsing context-free languages.

Memoization is a well-known technique in many programming languages, and tabled execution, including reduction operations, is well-known in logic programming. Some applications in functional programming have used techniques similar to tabled execution; see Section 6 for more information. A previous paper (Johnson 1995) explains tabled execution in Scheme in the context of language parsing; he also gives a progression from normal memoization to tabled execution using functions explicitly in CPS form. We use roughly the same model and sequence, with a somewhat different presentation; we also add the ability to combine the results of tabled functions, which does not appear in Johnson's paper. A more complete comparison of our work to his is given in Section 6. The example of paths within graphs is a simplification of a previous shortest path example using tabled execution (Dietrich 1992).

Tabled execution is an interesting use of Scheme because it mixes the use of continuation-passing style (CPS)—and thus higher-order functions—with impure operations. CPS is used to allow a function to pass any number (including none at all) of results to its continuation by calling it multiple times, and allows a function to be resumed in the middle when a new result arrives from a recursive call. Impure operations are used to keep the hash tables of memoization results and continuations. Also, the ability for a Scheme function to transform one function into another (first-class functions) is used to allow the wrapping of a function, and dynamic typing is used to allow a hash table element to store a value optionally.

Note that this paper addresses tabled execution of Scheme functions, and thus only uses functions that accept fully ground arguments and return fully ground results. Explicit continuation-passing style is used for returning results to allow a single invocation of a function to return multiple times (by calling its continuation repeatedly), but results cannot be wildcards as they can in a logic programming language. This model better matches the applications of tabled execution in the Scheme context, and removes the complexities of keeping tables with partially-unbound arguments; our implementation is also limited to a single scheduling strategy. Our system has not been tested with sophisticated uses of Scheme continuations in tabled functions, and likely will not work when they are used. The XSB Prolog manual presents more information on these design tradeoffs (Swift et al. 2007, §5.2).

We first review the basic idea of memoization, and show a simple memoization wrapper in Scheme (Section 2). We then change

* LLNL-CONF-406444. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded by the Laboratory Directed Research and Development Program at LLNL under project tracking code 07-ERD-057.

2008 Workshop on Scheme and Functional Programming

that wrapper to work on functions in continuation-passing style, including those that call their continuations multiple times (Section 3). We then discuss the problem of infinite recursions in multiple-return functions, and how tabled execution provides an elegant solution to that problem (Section 4). We then generalize our implementation of tabling to allow the results from a single function call to be combined in a user-defined way, rather than storing and returning all of them (Section 5). We then compare our approach to similar approaches used for related problems in Scheme, and discuss some previous work on tabled execution in Scheme (Section 6). A larger example, based on formal language and automata theory, is then presented (Section 7). We finish with a discussion of programming language issues in relation to memoization and tabling (Section 8) and then conclude (Section 9).

2. Memoization

Memoization is a standard technique for increasing the efficiency of computations (Michie 1968). A memoized function, instead of computing its result from scratch for every invocation, reuses its previous outputs when applied to the same arguments. A function can only be memoized if it is pure—it must always return the same output given the same inputs. A function that has been memoized keeps a cache of its previous arguments and results. If a new set of arguments matches a cached set, the cached results are used. Otherwise, the function is computed on the new set of arguments, and the result of that computation is inserted into the cache and then returned. An example of memoization is the following Fibonacci function implementation, which would require exponential time without memoization but requires only linear time with it:[1]

```
(define fib
 (let ((memo-table (make-hash-table)))
  (λ (n)
   (let ((memoized-val
          (hash-table-get memo-table n #f)))
    (if (eq? memoized-val #f)
     (let ((result
            (if (<= n 1)
                1
                (+ (fib (- n 1)) (fib (- n 2))))))
      (hash-table-put! memo-table n result)
      result)
     memoized-val)))))
```

Rather than converting a function to be memoized by hand, however, we can define a wrapper, as shown in Figure 1. Many others have devised such wrappers and automatic memoization translators in the past; we only present versions here to demonstrate them and provide a progression to the later demonstration of tabled execution. The `memoize` function's argument must be one-argument function.[2]

With this wrapper, `fib` can be written as follows (note that recursive calls must go through the memoization code to achieve linear execution time):

```
(define fib
 (memoize
  (λ (n)
   (if (<= n 1)
    1 (+ (fib (- n 1)) (fib (- n 2)))))))
```

[1] All Scheme code in this paper is in the PLT Scheme dialect.

[2] All memoization and tabled execution wrappers in this paper assume single-argument, single-result functions and may require that the function not return `#f` to allow its use as a sentinel value. The function parameters must be reliably comparable with `eq?` for brevity; calls to `make-hash-table` can be changed to allow `equal?` to be used instead.

```
(define memoize
 (λ (f)
  (let ((memo (make-hash-table)))
   (λ (arg)
    (let ((memo-value
           (hash-table-get memo arg #f)))
     (if (eq? memo-value #f)
      (let ((result (f arg)))
       (hash-table-put! memo arg result)
       result)
      memo-value))))))
```

Figure 1. Function memoization wrapper.

This new version of the code is much simpler and easier to understand, and requires only minimal modifications to the non-memoized version of `fib`.

3. CPS memoization

Given a simple memoization implementation, we can change it to operate on functions in continuation-passing style (CPS) (Steele 1978). The same basic components are used, except that an explicit continuation is used instead of `let` and normal function returns. CPS allows functions that produce a sequence of values for a single invocation by calling their continuations multiple times, and so we save all arguments to continuation calls. The new values are kept as a set (using a nested hash table), so that the multiple returns of the same value from the function are ignored. The memoization wrapper for this is:

```
(define cps-memoize-multi
 (λ (f)
  (let ((memo (make-hash-table)))
   (λ (k arg)
    (let ((memo-values
           (hash-table-get memo arg #f)))
     (if (eq? memo-values #f)
      (let ((output-values (make-hash-table)))
       (f (λ (result)
            (hash-table-put! output-values
                             result #t))
          arg)
       (hash-table-put! memo arg output-values)
       (hash-table-for-each output-values
                            (λ (v _) (k v))))
      (hash-table-for-each memo-values
                           (λ (v _) (k v)))))))))
```

This memoization wrapper can be used to compute all vertices in a directed acyclic graph (DAG) reachable from a given vertex:[3]

```
(define dag-reachable-vertices
 (cps-memoize-multi
  (λ (k v)
   (k v)
   (for-each
    (λ (tgt) (dag-reachable-vertices k tgt))
    (out-neighbors v)))))
```

This algorithm can then be called as:

```
(define out-neighbors
  (λ (v) (if (= v 9) '() (list (add1 v)))))
```

[3] The `out-neighbors` function is assumed to be global in order to avoid an extra `letrec` in the function definition.

```
(define display-space
 (λ (v) (display v) (display "␣")))
(dag-reachable-vertices display-space 0)

⟹ 1 3 9 5 7 8 6 4 2 0
```

4. Basic tabled execution

This implementation still has a problem: graphs with cycles cannot be used. The reason is that the `dag-reachable-vertices` function is defined recursively, and the recursion is only on the vertices of the graph. Thus, a vertex that is in a cycle will lead to an infinite recursion. Standard solutions to this problem involve passing around (or keeping in a mutable variable) state that indicates which vertices have previously been visited in the graph in order to avoid visiting them again. It would be better, however, to keep the simple interface of only having a single non-continuation argument to the reachable vertex functions.

The logic programming community has a solution to this problem, requiring only minimal changes to the `cps-memoize-multi` function. The technique is referred to as *tabled execution* (Warren 1992). It generalizes memoization by allowing the function to call itself recursively, even with the same arguments as in the current call. The function can even use results from such "infinite" recursions to produce its other results. In Scheme, some of the complexities of tabling in logic programming are unnecessary, as arguments to Scheme functions, and their results, are fully defined (ground). Using an implementation of this simplified form of tabled execution, a full graph reachability function can be defined exactly like the memoized definition for acyclic graphs, by only replacing the memoization wrapper:

```
(define graph-reachable-vertices
 (cps-table
  (λ (k v)
   (k v)
   (for-each
    (λ (tgt) (graph-reachable-vertices k tgt))
    (out-neighbors v)))))
```

The `cps-table` function is slightly trickier than the standard memoizer, however. First of all, it relies on CPS form for the functions; our functions already have that form. It also keeps a table of all of the continuations passed into it, in order to call them on any new answers produced by the function. The previous memoizer had no need to store the input continuations, because the function would never produce any new answers once it terminated; with this version, the function might initially reach an infinite recursion that is broken by the tabled execution implementation. If the function later produced an answer (using a different execution path), that answer would be inserted into the location of the infinite recursion. The function could then continue from that point to possibly produce more answers. The code is shown in Figure 2.

The `cps-table` function keeps two state values for each input argument value: the set `memo` (implemented using a hash table of hash tables) of answers previously returned by `f`, and the list `k-list` of continuations previously passed to the tabled version of `f`. The goal of a function `(cps-table f)` produced by `table-thunk` applied to `f` is to, when called with a continuation `k`, pass all answers produced by `f` to `k`. Thus, we want to generate the Cartesian product of the set of results from `f` with the set of continuations ever given to `(cps-table f)`, which can be viewed as a rectangle. However, both of these sets are dynamic: calls to `(cps-table f)` can occur at any time, including in the body of `f` (adding more continuations to `k-list`); and `f` can return answers at any time (adding more members to `memo`). These operations can occur in any order, and each new element of either set is processed using all elements so far of the other set, increasing the size of one dimension of the rectangle; duplicate results from `f` are skipped by the check in lines 14–16. Lines 20–22 call all continuations for a new result from `f`, and lines 23–25 call all new continuations given to `(table-thunk f)` with all results so far. Lines 6–9 and 18–19 update the total sets of continuations and results, respectively. The strategy for processing the elements of the rectangle can be varied. The other necessary behavior of the code is to only start `f` once, no matter how many times `(table-thunk f)` is called; this check is done on line 10 using the value `#f` in `memo` to indicate that `f` has not yet been started.

```
1 (define cps-table
2  (λ (f)
3   (let ((memo (make-hash-table))
4         (k-list (make-hash-table)))
5     (λ (k arg)
6      (hash-table-put!
7       k-list
8       arg
9       (cons k (hash-table-get k-list arg '())))
10     (if (eq? (hash-table-get memo arg #f) #f)
11      (let ((memo-table (make-hash-table)))
12       (hash-table-put! memo arg memo-table)
13       (f (λ (result)
14           (if (eq? (hash-table-get memo-table
15                                    result #f)
16                    #f)
17            (begin
18             (hash-table-put! memo-table
19                              result #t)
20             (for-each
21              (λ (saved-k) (saved-k result))
22              (hash-table-get k-list arg '()))))
23          arg))
24       (hash-table-for-each
25        (hash-table-get memo arg)
26        (λ (v _) (k v))))))))
```

Figure 2. The simple tabled execution wrapper.

5. Tabled execution with reduction operators

Instead of computing the reachability, suppose that the goal is to compute single-source shortest paths in a weighted graph, as is done with tabled execution in (Dietrich 1992). To get the paths, the best path length associated with each reachable vertex needs to be stored, not all of them. The memo table for a function and a given argument is now a map rather than a set; we assume for simplicity that `#f` is not a legal length value. The code is changed to both handle two-argument continuations and to only keep the shortest length value for each output vertex. Also, note that function outputs that are not new length minima are completely ignored; they are not passed to the output continuations. This modified version of the code is in Figure 3.

This code only works correctly as long as the uses of this code are monotonic in the length value; only new minimal lengths are passed to the continuations, and multiple lengths (not just the final minimum for each vertex) are given to them. Shortest path computation works correctly with these issues, but other algorithms might need to be modified to support being called with non-final values.

```
(define cps-table-min
 (λ (f)
  (let ((memo (make-hash-table))
        (k-list (make-hash-table)))
    (λ (k arg)
     (hash-table-put!
      k-list
      arg
      (cons k (hash-table-get k-list arg '())))
     (if (eq? (hash-table-get memo arg #f) #f)
      (let ((memo-table (make-hash-table)))
       (hash-table-put! memo arg memo-table)
       (f (λ (result len)
           (let ((old-len
                  (hash-table-get memo-table
                                  result #f)))
            (if (or (eq? old-len #f)
                    (< len old-len))
             (begin
              (hash-table-put! memo-table
                               result len)
              (for-each
               (λ (saved-k) (saved-k result len))
               (hash-table-get k-list arg '()))))))
          arg))
      (hash-table-for-each
       (hash-table-get memo arg)
       k))))))
```

Figure 3. A tabled execution wrapper for functions that return pairs $\langle result, length \rangle$ and where only the minimum *length* for each *result* is kept.

```
(define cps-table-combine
 (λ (f combine)
  (let ((memo (make-hash-table))
        (k-list (make-hash-table)))
   (λ (k arg)
    (hash-table-put!
     k-list arg
     (cons k (hash-table-get k-list arg '())))
    (if (eq? (hash-table-get memo arg #f) #f)
     (let ((memo-table (make-hash-table)))
      (hash-table-put! memo arg memo-table)
      (f (λ (result len)
          (let* ((old-len
                  (hash-table-get memo-table
                                  result #f))
                 (new-len (combine old-len len)))
           (if (not (equal? old-len new-len))
            (begin
             (hash-table-put! memo-table
                              result new-len)
             (for-each
              (λ (saved-k) (saved-k result len))
              (hash-table-get k-list arg '()))))))
         arg))
     (hash-table-for-each (hash-table-get memo arg)
                          k))))))
```

Figure 4. A wrapper for tabled functions that return $\langle result, length \rangle$ pairs, allowing the *length* values for a single *result* to be combined using a user-defined operator.

There is a further improvement that could be made to this code: generalizing the min operation to an arbitrary user-defined combination function. The previous code sample used an explicit `<` operation combined with a conditional update of the hash value for that operation, but a more general function will need to be given both the old and proposed new results of the function for a given argument set. Also, rather than conditionally calling the continuations based on the result of the `<` operation, an explicit check for the equality of the old and new "length" values for each result must be done. The `combine` function is required to be idempotent and have `#f` as an identity element (which may be provided in a wrapper if `#f` is not ordinarily in the domain of the function). This generalization leads to the code in Figure 4.

This version of the code still requires the values produced by `f` to be $\langle value, length \rangle$ pairs, where *length* might be some length-like object that is combined using the `combine` function. Functions that only return single values rather than pairs can be changed to functions that, say, return `#f` and their actual output. This approach is inefficient, but the code could be partially evaluated for that special case. The code also does not allow changes in the calling sequence for `k` and elements in `k-list`. Thus, this function could be generalized further to allow a function to be used to call continuations. This new feature would allow continuations to be called with multiple arguments, or to not be functions at all. The only change that would be required to the code is to add a `call-k` parameter, and to use it whenever a continuation is invoked rather than calling it directly. That version is a straightforward modification of the code above and so is not shown.

One difference between the code shown in this section and the programming style shown in (Warren 1999) is that our implementation marks particular functions as being tabled with reduction operations, while their implementation implements higher-order query predicates on existing tables. We can simulate that model using a second table which copies an existing, non-combining table and applies a combination operator to it; thus, this difference in implementations does not cause a problem.

6. Comparison with related work

Tabled execution, and constructs similar to it, have appeared a few times in Scheme. A paper by Johnson (1995) uses tabled execution, in a similar form to that in this paper, for top-down parsing. He, however, also defines a general-purpose tabled execution wrapper for Scheme functions in CPS form. He also gives a tutorial introduction to tabled execution for use in parsing which starts with normal and then CPS memoization, and then progresses to tabled execution. However, his paper does not include operators for combining multiple results from a single call to a tabled function. He also uses lists and a subsumption check, rather than a hash table, to store multiple results from one function with one set of arguments.

A number of other control flow techniques used in Scheme can be confused with tabled execution. A comparison of their features is given in Table 1. Normal and CPS memoization have already been explained, as have the forms of tabled execution. Some systems use normal memoization with an explicit check for infinite recursion. For example, they may pass a stack of currently active calls as a parameter to their recursive calls, and check new arguments against the stack; the function returns a special value if the current argument is already on the recursion stack. Another approach to the same problem is to keep a hash table of arguments that are either in progress or have been done already; graph node reachability can be solved using this technique. However, it is not as powerful as tabled execution. For example, tabled execution allows the following implementation of `graph-reachable-vertices` which would produce incorrect behavior using simpler techniques:

Technique	Structure req'd	Multiple answers	Prevents loops	Answers can be looped	Fixpoint ops
Normal memoization	Normal	N/A			
CPS memoization	CPS	✓			
Memoization with recursion checking	Normal	N/A	✓		
Basic tabled execution	CPS	✓	✓	✓	
Tabled execution with combine operators	CPS	Maybe	✓	✓	
Explicit fixpoint evaluation	Usually custom	Maybe	✓	✓	✓

Table 1. Techniques for memoization of function results; "multiple answers" refers to passing answers back to the caller through multiple calls to a continuation, "looping" refers to infinite recursions with the same arguments, and "answers can be looped" refers to returning answers produced by outer calls as results of inner looped calls (allowing the reachability example in Section 6 to work correctly).

```
(define graph-reachable-vertices
 (cps-table
  (λ (k v)
   (k v)
   (graph-reachable-vertices
    (λ (tgt) (for-each k (out-neighbors tgt)))
    v))))
```

This code does a recursion on the vertex `v` before the call to `out-neighbors` rather than after it. A simple check for repeated vertices would not work for this code, but tabled execution produces correct behavior. The difference between simple loop checking and tabled execution is that tabled execution does not just ignore infinite recursions, it suspends them and passes answers from the outer call as results from the inner one. The grammar example in Section 7 also requires tabled execution: a grammar rule may have multiple nonterminals on its right-hand side, and any of them could be recursive. Left-recursive grammars, in particular, require that automaton states from an inner recursive call be passed back to the outer call to be processed.

Another alternative to tabled execution is to implement an explicit fixpoint computation. This approach has the disadvantage that it is more complicated, but it can express a greater variety of computations. Tabled execution is effectively finding the fixpoint of a system of set equations, but has limits with more general equations. An explicit fixpoint solver can do optimizations that are not available to systems that keep control flow between tabled functions implicit. However, one issue with explicit fixpoint solving is that it is not usually just a wrapper around normal functions, although it can be implemented that way. It does, however, trigger the issues described in Section 8, making implementation more difficult.

All of the features in this paper are from previous tabled execution literature using Prolog and other similar logic programming languages. The uses of tabled execution for analyzing context-free languages and for graph algorithms are also from that literature.

7. Verifying context-free grammars using finite-state automata

A larger example that better motivates the use of tabled execution is the problem of finding the result states of a finite-state automaton when run on all strings generated by a context-free grammar. That is, given a context-free grammar G with start nonterminal S and a nondeterministic finite-state automaton A with start state q, the goal is to find all states q' such that there is a string w in the language of G such that $q \xrightarrow{w} q'$. This problem is related to program verification: G is the grammar of possible execution traces of a program (allowing context-sensitive handling of procedures), and A is the automaton of traces that satisfy the desired property; determining the states of A reached after all runs of the program and checking those against A's accepting states determines whether the program always, sometimes, or never has the property. Such a model is used in, among others, MOPS (Chen and Wagner 2002) and FLAVERS (Dwyer et al. 2004). The transition function Δ of A is represented as a function, rather than a table, because the set Q of states of A might be large, or even infinite; of course, the set of reachable states over a given context-free grammar must be finite for the algorithm to terminate. Similarly, we do not assume the ability to enumerate Q as would standardly be done for the conversion of a push-down automaton to a context-free grammar.

In Scheme, the grammar can be represented by a function (in CPS) that accepts a nonterminal name and a continuation, and calls the continuation once for each grammar rule with that nonterminal as its left-hand side. A symbol in the grammar (either a terminal or nonterminal) is represented as an element of the set Γ, with terminals in Σ and nonterminals in $\Gamma - \Sigma$. The argument to the continuation is a list of terminals (non-symbols) and nonterminals (symbols) for the rule:

$$G : ([\Gamma] \to \mathbf{1}) \times (\Gamma - \Sigma) \to \mathbf{1}$$

The start nonterminal is given separately. Similarly, the automaton can be represented as a function Δ (again in CPS) that takes a state from the state set Q and a symbol from the alphabet Σ and returns (through multiple continuation calls) a set of new states:

$$\Delta : (Q \to \mathbf{1}) \times Q \times \Sigma \longrightarrow \mathbf{1}$$

Again, the start state is given separately. Now, the goal is to write a function `possible-states` that maps a nonterminal or terminal from the grammar and a state from the automaton into a set of resulting states from the automaton, as well as the lengths of the strings producing each output state. As before, the function will pass multiple values to its continuation:

$$\texttt{possible-states} : (Q \times \mathbb{N} \to \mathbf{1}) \times \Gamma \times Q \longrightarrow \mathbf{1}$$

First, a basic implementation of `possible-states` is defined for acyclic grammars is shown in Figure 5. This code fails for most grammars that contain recursion, however: if `grammar-elt` and `state` are the same in nested calls to `possible-states`, an infinite loop results. Tabled execution solves this problem. Wrapping `possible-states` for tabled execution using the previously defined wrapper (modified for two-argument functions), as shown in Figure 6, solves this problem by preventing infinite recursions while preserving correct behavior. Note that `possible-states*` cannot recur forever except through `possible-states` as the lists of symbols in a grammar rule must be finite; thus, only one function needs to be wrapped.

The only change is to wrap `cps-table-min` around the function. Note that recursions must be to the wrapped function, rather than the inner body; assuming that `possible-states*` refers to the name `possible-states` rather than its body, that will happen automatically. This new function also only keeps the length of the shortest string producing each output state rather than all of them. Generalizing the code to return the strings themselves requires switching to `cps-table-combine`, an appropriate reduction function, and small changes to the rest of the code. This generalization would be able to show the user a short example program execution that violates the correctness property being checked.

```
(define possible-states ; Run on one symbol
 (λ (k grammar-elt state)
  (if (symbol? grammar-elt) ; Nonterminal?
   (G (λ (syms)
        (possible-states* k syms state))
      grammar-elt)
   (Delta (λ (state2) (k state2 1))
          state grammar-elt))))

(define possible-states* ; Run on list of symbols
 (λ (k grammar-elt* state)
  (if (null? grammar-elt*)
   (k state 0)
   (possible-states
    (λ (state2 len2)
     (possible-states*
      (λ (state3 len3) (k state3 (+ len2 len3)))
      (cdr grammar-elt*)
      state2))
    (car grammar-elt*)
    state))))
```

Figure 5. The `possible-states` function for acyclic grammars, and the `possible-states*` function.

```
(define possible-states ; Run on one symbol
 (cps-table-min
  (λ (k grammar-elt state)
   (if (symbol? grammar-elt) ; Nonterminal?
    (G (λ (syms)
         (possible-states* k syms state))
       grammar-elt)
   (Delta (λ (state2) (k state2 1))
          state grammar-elt)))))
```

Figure 6. The `possible-states` function for arbitrary context-free grammars.

8. Language design issues

Although the implementation of tabled execution for Scheme would apply to any functional language with mutable hash tables or boxes, and less transparently if a monad is required for side effects, it has some limitations that could be solved by language modifications. Procedures represented by explicit data structures (closures) can be compared deeply for equality, while standard Scheme procedures can only be compared shallowly. This section shows how that can cause problems for memoization, and therefore tabled execution. One practical example of such a problem is the automaton and grammar handling in Section 7: the states of the automaton A cannot be represented as functions unless they are not parameterized; it is not possible to have, for example, an automaton whose states are predicates on alphabet symbols. Such a capability would make certain automata much more elegant to write.

One major problem with the given implementation, even the basic memoization wrapper, is that higher-order functional programming is not supported. In particular, currying a tabled procedure does not work straightforwardly. Assuming a two-argument function `f`, it is possible to produce a curried, memoized version of `f` as `(let ((m (memoize f))) (λ (x) (λ (y) (m x y))))`, but other, normally equivalent forms of currying do not work when combined with memoization. The "obvious" currying of `(memoize f)`, `(λ (x) (λ (y) ((memoize f) x y)))` does not result in any memoization at all: `(memoize f)` is re-evaluated for each set of arguments, and so the same memoized function is never used twice. It is also not possible to implement the memoization as `(memoize (λ (x) (λ (y) (f x y))))`: even though that does save the closure `(λ (y) (f x y))` for each value of `x`, the lack of memoization in the nested closures means that nothing is actually saved. Using `(λ (x) (memoize (λ (y) (f x y))))` fails for a different reason: the call to `memoize` is repeated for each call to the function, and so each memoized function is discarded immediately after use. A correct modification of these two versions is `(memoize (λ (x) (memoize (λ (y) (f x y)))))`, in which both levels of memoization are necessary: the first ensures that the second `memoize` call is only run once for each distinct value of `x`. The main point of this demonstration is to show that users must be careful when mixing memoization or tabling with higher-order functions: it is easy to make mistakes that make memoization useless, or make tabled operations fail to terminate because they do not recognize that a particular call forms part of an infinite loop.

A similar issue arises with memoized higher-order functions, such as `(λ (x ls) (map (λ (y) (< x y)) ls))`. This code does not benefit from a memoized version of `map`: the closure argument is never repeated. The `(λ (y) (< x y))` closure for each value of `x` would need to be created by a memoized procedure so that exactly the same function is given to `map` for each value of `x`.

The issue that leads to all of these subtleties is the fact that most versions of Scheme, as well as other functional languages, do not allow the deep comparison of closures for equality. For example, if the function `(λ (x) (λ () x))` is evaluated at two different times with the same value of `x`, the results will be completely different procedures that do not compare equal to each other, even using `equal?`. In general, if `f` returns a closure that uses `x`, `(f x)` is not `equal?` to `(f x)` called at a different time, even when the same values of `f` and `x` are used. Mathematically, of course, it does not make sense to try to compare functions too deeply; it is undecidable to determine whether two functions are extensionally equal. However, a deeper level of comparison than the default `eq?` (used by most implementations for `equal?` on procedures) could make memoization and tabling more effective. Some Scheme implementations already provide deeper comparisons: PLT Scheme version 3 allows a closure comparison (`procedure-closure-contents-eq?`, mentioned in its release notes (Flatt 2006)), and Guile provides a function to determine the environment of a closure (`procedure-environment`, documented in (Guile Developers 2005, §5.8.4) and used for closure comparison in (Marton 2008)). Implementations of closures in compilers and interpreters generally use a data structure that allows comparison; it is just not exposed to users. Explicit conversion of procedures to data structures by the user allows these transformed closures to be compared, but this conversion is not compositional (requiring modifications to all procedures used at a call site) and negates the benefits of using higher-order procedures in the first place.

9. Conclusion

This Scheme Pearl shows how to apply the tabled execution technique from logic programming in the context of Scheme. It demonstrates how tabled execution is a generalization of memoization, and that continuation-passing style makes tabled execution simple to implement. It combines the use of higher-order functions with impure operations on hash tables to achieve this more sophisticated form of memoization, and relies on CPS and multiple calls to the same continuation to operate. An example then shows that tabled execution can elegantly solve a real-world problem.

This work could be extended in several directions. One would be to create a more elegant wrapper for tabled execution with combine operators. Such a wrapper could be given a combine operation as input, as in the version in Section 5. Calls to continuations in that code could be generalized. For example, each continuation `k` could be called with all results so far, the combined result so far, or something completely different (for example, the difference between the previous combined result and the new one, as is done in the FLAVERS system (Dwyer et al. 2004)).

A similar approach to our implementation of tabled execution could also be used to define systems of equations over lattices and determine their fixpoints, as is required for program analysis. Each variable is represented using its current value, a function that computes a new value based on the values of other variables in the system of equations, and a list of its dependencies on other variables. A worklist algorithm can then be used to iteratively update the variables when their dependencies change until a fixpoint is reached. An elegant implementation of this algorithm, as opposed to a straightforward implementation as used in imperative languages, remains as future work; tabled execution with combine operators might provide a basis. We have created a basic version of fixpoint solving using a simple wrapper function, but the issues described in Section 8 make it inelegant to use.

It might also be interesting to combine the tabled execution system with a logic programming engine embedded into Scheme, such as Kanren (Friedman et al. 2005). This integration would require the correct handling of partially instantiated values. Tabled Prolog could then be implemented, allowing a logically complete, terminating Datalog implementation to be embedded in Scheme.

Acknowledgments

The first author was partially supported by a United States Department of Energy High Performance Computer Science Fellowship. This project was also partially funded by the Air Force Research Laboratory, as well as by NSF award CCF-0541335 and the Lilly Endowment. We would also like to thank Daniel P. Friedman for helpful discussions and guidance, and Chunhau Liao and Thomas Panas for their comments on the paper; the anonymous reviewers also gave many good suggestions.

References

Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-612-9.

Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996. ISSN 0004-5411.

Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *Programming Language Design and Implementation*, pages 117–126, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-795-2.

Suzanne W. Dietrich. Shortest path by approximation in logic programs. *ACM Letters on Programming Languages and Systems*, 1(2):119–137, 1992. ISSN 1057-4514.

Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol.*, 13(4):359–430, 2004. ISSN 1049-331X.

Changguan Fan and Suzanne Wagner Dietrich. Extension table built-ins for Prolog. *Software—Practice and Experience*, 22(7): 573–597, 1992. ISSN 0038-0644.

Matthew Flatt. 301.11. Mailing list posting, March 2006. `http://www.cs.brown.edu/pipermail/plt-scheme/2006-March/012289.html`.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, 2005. ISBN 0262562146.

Guile Developers. *The Guile Reference Manual*. Free Software Foundation, 1.1 edition, 2005. `http://www.gnu.org/software/guile/manual/`.

Mark Johnson. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–417, 1995. `http://www.aclweb.org/anthology-new/J/J95/J95-3005.pdf`.

Gregory A. Marton. comparing procedures. Mailing list posting, January 2008. `http://www.mail-archive.com/bug-guile@gnu.org/msg04254.html`.

Donald Michie. "Memo" functions and machine learning. *Nature*, 218(5136):19–22, April 1968. URL `http://dx.doi.org/10.1038/218019a0`.

Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David Scott Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification*, pages 143–154, London, UK, 1997. Springer-Verlag. ISBN 3-540-63166-6.

Diptikalyan Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Declarative Programming*, pages 117–128, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-090-6.

Guy L. Steele, Jr. Rabbit: A compiler for Scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

Terrance Swift et al. *The XSB System, Volume 1: Programmer's Manual*, version 3.1 edition, August 2007. `http://xsb.sourceforge.net/manual1/`.

Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98, London, UK, 1986. Springer-Verlag. ISBN 3-540-16492-8.

David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992. ISSN 0001-0782.

David S. Warren. *Programming in Tabled Prolog*. Unpublished, July 1999. Draft version at `http://www.cs.sunysb.edu/~warren/xsbbook/book.html`.

David Scott Warren. Programming the PTQ grammar in XSB. In *Workshop on Programming with Logic Databases (Book), ILPS*, pages 217–234, 1993. `ftp://ftp.cs.sunysb.edu/pub/XSB/doc/XSB/ptq_in_xsb.ps.gz`.