Efficient tabling of structured data with enhanced hash-consing

Zhou, Neng-Fa; Have, Christian Theil

Published in:
Theory and Practice of Logic Programming

DOI:
10.1017/S1471068412000178

Publication date:
2012

Document Version
Early version, also known as pre-print

Citation for published version (APA):
Efficient Tabling of Structured Data with Enhanced Hash-Consing

Neng-Fa Zhou
CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu

Christian Theil Have
Roskilde University
cth@ruc.dk

submitted; revised; accepted

Abstract

Current tabling systems suffer from an increase in space complexity, time complexity or both when dealing with sequences due to the use of data structures for tabled subgoals and answers and the need to copy terms into and from the table area. This symptom can be seen in not only B-Prolog, which uses hash tables, but also systems that use tries such as XSB and YAP. In this paper, we apply hash-consing to tabling structured data in B-Prolog. While hash-consing can reduce the space consumption when sharing is effective, it does not change the time complexity. We enhance hash-consing with two techniques, called input sharing and hash code memoization, for reducing the time complexity by avoiding computing hash codes for certain terms. The improved system is able to eliminate the extra linear factor in the old system for processing sequences, thus significantly enhancing the scalability of applications such as language parsing and bio-sequence analysis applications. We confirm this improvement with experimental results.

1 Introduction

Tabling, as provided in logic programming systems such as B-Prolog (Zhou et al. 2008), XSB (Swift and Warren 2012), YAP (Santos Costa et al. 2012), and Mercury (Somogyi and Sagonas 2006), has been shown to be a viable declarative language construct for describing dynamic programming solutions for various kinds of real-world applications, ranging from program analysis, parsing, deductive databases, theorem proving, model checking, to logic-based probabilistic learning. The main idea of tabling is to memorize the answers to subgoals in a table area and use the answers to resolve their variant or subsumed descendants. This idea of caching previously calculated solutions, called memoization, was first used to speed up the evaluation of functions (Michie 1968). Tabling can get rid of not only infinite loops for bounded-term-size programs but also redundant computations in the execution of recursive programs. While Datalog programs require tabling only subgoals with atomic arguments, many other programs such as those dealing with complex language corpora or bio-sequences require tabling structured data. Unfortunately, none
N.F. Zhou and C. T. Have

of the current tabling systems can process structured data satisfactorily. Consider, for example, the predicate \texttt{is_list/2}:

\begin{verbatim}
:-table is_list/1.
  is_list([]).
  is_list([_|L]):-is_list(L).
\end{verbatim}

For the subgoal \texttt{is_list([1,2,...,N])}, the current tabled Prolog systems demonstrate a higher complexity than linear in \( N \): B-Prolog (version 7.6 and older) consumes linear space but quadratic time; YAP, with a global trie for all tabled structured terms (Raimundo and Rocha 2011), consumes linear space but quadratic time; XSB is quadratic in both time and space. The nonlinear complexity is due to the data structure used to represent tabled subgoals and answers and the need to copy terms into and from the table area.

The inefficiency of early versions of B-Prolog in handling large sequences has been reported and a program transformation method has been proposed to index ground structured data to work around the problem (Have and Christiansen 2012). In old versions of B-Prolog, tabled subgoals and answers were organized as hash tables, and input sharing was exploited to allow a tabled subgoal to share its ground structured arguments with its answers and its descendant subgoals. Input sharing enabled B-Prolog to consume only linear space for the tabled subgoal \texttt{is_list([1,2,...,N])}. Nevertheless, since the hash code was based on the first three elements of a list, the time complexity for a query like \texttt{is_list([1,1,...,1])} was quadratic in the length of the list. B-Prolog didn’t support output sharing, i.e. letting different answers share structured data. Therefore, on the tabled version of the permutation program that generates all permutations through backtracking, B-Prolog would create \( n \times n! \) cons cells where \( n \) is the length of the given list.

This problem with tabling structured data has been noticed before and several remedies have been attempted. One well known technique used in parsing is to represent sentences as position indexed facts rather than lists. XSB provides tabled grammar predicates that convert list representation to position representation by redefining the built-in predicate \texttt{\textsc{C}/3}.\footnote{Personal communication with David S. Warren, 2011.} The position representation is also used for PCFG parsing in PRISM (Sato and Kameya 2008). A program transformation method has been proposed to index ground structured data to work around the quadratic time complexity of B-Prolog’s tabling system (Have and Christiansen 2012). Nevertheless, these remedies have their limitations: the position representation disallows natural declarative modeling of sequences and the program transformation incurs considerable overhead. Have and Christiansen advocate for native support of data sharing in tabled Prolog systems for better scalability of their bio-sequence analysis application (Have and Christiansen 2012).

We have implemented full data sharing in B-Prolog in response to the manifesto. In the new version of B-Prolog, both input sharing and output sharing are exploited to allow tabled subgoals and answers to share ground structured data. \textit{Hash-consing}
Efficient Tabling of Structured Data with Enhanced Hash-Consing

(Ershov 1959), a technique originally used in functional programming to share values that are structurally equal (Goto 1974; Appel and de Rezende Goncalves 2003), is adopted to memorize structured data in the table area. This technique avoids storing the same ground term more than once in the table area. While hash-consing can reduce the space consumption when sharing is effective, it does not change the time complexity. To avoid the extra linear time factor in dealing with sequences, we enhance hash-consing with input sharing and hash code memoization. For each compound term, an extra cell is used to store its hash code. Thanks to these techniques, B-Prolog demonstrates linear complexity in terms of both space and time on the query \texttt{is\_list}(L) for any kind of ground list L.

Our main contribution in this paper is to apply hash-consing for the first time to tabling and enhance it with techniques to make it time efficient. As another contribution, we also compare tries with hash consing in the tabling context. As long as sequences are concerned, a trie allows for sharing of prefixes while hash-consing allows for sharing of ground suffixes. While we can build examples that arbitrarily favor one over the other, for recursively defined predicates such as \texttt{is\_list}, it is more common for subgoals to share suffixes than prefixes. The enhanced hash-consing greatly improves the scalability of PRISM on sequence analysis applications. Our experimental results on a simulator of a hidden Markov model show that PRISM with enhanced hash-consing is asymptotically better than the previous version that supports no hash-consing.

The remainder of the paper is structured as follows: Section 2 defines the primitive operations on the table area used in a typical tabling system; Section 3 presents the hash tables for subgoals and answers, and describes the copy algorithm for copying data from the stack/heap to the table area; Section 4 modifies the copy algorithm to accommodate hash-consing; Section 5 describes the techniques for speeding up computation of hash codes; Section 6 evaluates the new tabling system with enhanced hash-consing; Section 7 gives a survey of related work; and Section 8 concludes the paper.

2 Operations on the Table Area

A tabling system uses a data area, called \textit{table area}, to store tabled subgoals and their answers. A tabling system, whether it is suspension-based SLG (Chen and Warren 1996) or iteration-based linear tabling (Zhou et al. 2008), relies on the following three primitive operations to access and update the table area.\(^2\)

\textbf{Subgoal lookup and registration:} This operation is used when a tabled subgoal is encountered in execution. It looks up the subgoal table to see if there is a variant of the subgoal. If not, it inserts the subgoal (termed a \textit{pioneer} or \textit{generator}) into the subgoal table. It also allocates an answer table for the subgoal and its variants. Initially, the answer table is empty. If the lookup finds that there already is a variant of the subgoal in the table, then the record stored in the table is used

\(^2\) The interpretation of these operations may vary depending on implementations.
for the subgoal (called a consumer). Generators and consumers are dealt with differently. In linear tabling, for example, a generator is resolved using clauses and a consumer is resolved using answers; a generator is iterated until the fixed point is reached and a consumer fails after it exhausts all the existing answers.

**Answer lookup and registration:** This operation is executed when a clause succeeds in generating an answer for a tabled subgoal. If a variant of the answer already exists in the table, it does nothing; otherwise, it inserts the answer into the answer table for the subgoal. When the lazy consumption strategy (also called local strategy) is used, a failure occurs no matter whether the answer is in the table or not, which drives the system to produce the next answer.

**Answer return:** When a consumer is encountered, an answer is returned immediately if any. On backtracking, the next answer is returned. A generator starts consuming its answers after it has exhausted all its clauses. Under the lazy consumption strategy, a top-most looping generator does not return any answer until it is complete.

### 3 Hash Tables for Subgoals and Answers

The data structures used for the table area are orthogonal to the tabling mechanism, whether it is suspension-based or iteration-based; they can be hash tables, tries, or some other data structures. In this section, we consider hash tables and the operations for the table area without data sharing.

A hash table, called a **subgoal table**, is used for all tabled subgoals. For each tabled subgoal and its variants, there is a record in the subgoal table, which includes, amongst others, the following fields:

- **AnswerTable**: Pointer to the answer table for the subgoal
- **sym**: The functor of the subgoal
- **A1...An**: The arguments the subgoal

When a tabled predicate is invoked by a subgoal, the subgoal table is looked up to see if a variant of the subgoal exists. If not, a record is allocated and the arguments are copied from the stack/heap to the table area. The copy of the subgoal shares no structured terms with the original subgoal and all of its variables are numbered so that they have different identities from those in the original subgoal.

The record of a subgoal in the subgoal table includes a pointer to another hash table, called an **answer table**, for storing answers produced for the subgoal. For each answer and its variants, there is a record in the answer table, which stores amongst others a pointer to a copy of the answer. When an answer is produced for a subgoal, the subgoal’s answer table is looked up to see if a variant of the answer exists. If not, a record is allocated and the answer is copied from the stack/heap to the table area. The answers in a subgoal’s answer table are connected from the oldest one to the newest one such that they can be consumed by the subgoal one by one through backtracking.

In the implementation, a hash table is represented as an array. To add an item
into a hash table, the system computes the hash code of the item and uses the hash code modulo the size of the array to determine a slot for the item. All items hashed to the same slot are connected as a linked list, called a hash chain. A hash table is expanded when the number of records in it exceeds the size of the array.

The WAM representation (Warren 1983) is used to represent both terms on the heap and terms in the table area except that variables in tabled terms are numbered. A term is represented by a word containing a value and a tag. The tag distinguishes the type of the term. It may be REF denoting a reference, ATM an atomic value, STR a structure, LST a cons, or NUMVAR a numbered variable. A STR-tagged reference to a structure $f(t_1, \ldots, t_n)$ points to a block of $n+1$ consecutive words where the first word points to the functor $f/n$ in the symbol table and the remaining $n$ words store the $n$ components of the structure. An LST-tagged reference to a list cons $[H|T]$ points to a block of two consecutive words where the first word stores the car $H$ and the second word stores the cdr $T$.

Figure 1 gives the definition of the function `copy_term` that copies a numbered term from the stack/heap to the table area. The hash function is designed in such a way that the hash code of a non-ground term is always 0. The function call `seq_hcode(code1, code2)` gives the combined hash code of the two hash codes from two components:

```c
int seq_hcode(int code1, int code2){
    if (code1==0) return 0;
    if (code2==0) return 0;
    return code1+31*code2+1;
}
```

If either code is 0, then the resulting code is 0 too.\(^3\)

It is assumed that all the variables in a subgoal have been numbered before the arguments are copied. In the real implementation, variables are numbered inside the function `copy_term`. The function call `copy_subgoal_args(src, des, arity)` copies the arguments of a numbered subgoal to the table area where $(src-i)$ points to the $i$th argument on the stack and $(des+i)$ is the destination in the table area where the argument is copied to. In the TOAM architecture (Zhou 2012) on which B-Prolog is based, arguments are passed through the stack and the stack grows downward from high addresses to low ones. That is why $(src-1)$ points to the first argument and $(src-arity)$ points to the last argument of the subgoal. A similar function is used to copy answers to the table area.

The function `copy_term` is not tail recursive and can easily cause the native C stack to overflow when copying large lists. In the real implementation, an iterative version is used to copy a list and compute its hash code. For a cons, the function needs to compute the hash codes of the car and the cdr before computing its hash code. The function does this in two passes: in the first pass it reverses the list and in the second pass it computes the hash codes while reversing the list back.

---

\(^3\) Note that this way of combing hash codes is for hash consing terms. For the subgoal and answer tables, hash codes are combined in a different way.
The function `copy_term` exploits no sharing of data. Consider, for example, the following program and the query \texttt{is\_list([1,2])}. After completion of the query, the subgoal table contains three tabled subgoals, \texttt{is\_list([1,2])}, \texttt{is\_list([2])}, and \texttt{is\_list([])}, and each subgoal’s answer table contains an answer that is just a copy of the subgoal itself. No data are shared among the copies of the terms. So there are two separate copies of \texttt{[1,2]} and two separate copies of \texttt{[2]} in the table area. In the WAM representation of lists, a cons requires two words to store, so 12 words are used in total. In general, the query \texttt{is\_list([1,2,\ldots,N])} consumes $O(N^2)$ space in the table area.

4 Hash-Consing of Ground Compound Terms

Hash-consing, like tabling, is a memoization technique which uses a hash table to memorize values that have been created. Before creating a new value, it looks up the table to see if the value exists. If so, it reuses the existing value, otherwise, it inserts the value into the table. The concept of hash-consing originates from implementations of Lisp that attempt to reuse cons cells that have been constructed.
Efficient Tabling of Structured Data with Enhanced Hash-Consing

This technique has also been suggested for Prolog (e.g., for sharing answers of findall/3 (O’Keefe 2001)), but its use in Prolog implementations is unknown, not to mention its use in tabling.

Let’s call the hash table used for all ground terms terms-table. Figure 2 gives an updated version of copy_term that performs hash-consing. If the term is a list or a structure, the function copies it into the table area first. If the term is ground, it then calls the function hash_consing(t1,hcode) to look up the terms-table to see if a copy of t1 already exists in the table. If so, hash_consing(t1,hcode) returns the copy; otherwise, it inserts t1 into the terms-table and returns t1 itself. If an old copy in the terms-table is returned (t1 != t2), the function deallocates the memory space allocated for the current copy.

With hash-consing, the query ?-is_list([1,2]) only creates one copy of [1,2] in the table area and the list is shared by the subgoals and the answers. As [2] is the cdr of [1,2], no separate copy is stored for it. So, only 4 words are used in total for the list. The number of words used for hashing the two lists varies, depending on if there is a collision. If no collision occurs, two slots in the terms-table are used; otherwise, one slot in the terms-table is used and one node with two words is used to chain the two lists. So in the worst case, 7 words are needed in total.

5 Enhanced Hash-Consing

With hash-consing, the tabled subgoal is_list([1,...,N]) consumes only linear table space now. Nevertheless, its time complexity remains quadratic in N. This is because for each descendant subgoal is_list([K,...,N]) (K>1) the hash code of the list [K,...,N] has to be computed and the terms-table has to be looked up. We enhance hash-consing with two techniques to lower the time complexity of is_list([1,...,N]) to linear.4

5.1 Hash code memoization

The first technique is to table hash codes of structured terms in the table area. For each structure or a list cons in the table area, we use an extra word to store its hash code. The WAM representation of terms is not changed. The word for the hash code of a compound term is located right before the term. So assume p is the untagged reference to a structure or a list cons, then p-1 references the hash code.

Figure 3 gives a new version of copy_term that tables hash codes. Tabled hash codes are used for two purposes. Firstly, when searching for the term t1 in the hash chain, the function hash_consing(t1,hcode) always compares the hash codes first and only when the codes are equal will it compare the terms. Secondly, the system reuses the tabled hash codes of terms when it expands a hash table and rehashes the terms into the new hash table.

With tabled hash codes, the subgoal is_list([1,...,N]) still takes quadratic

4 The worst case time complexity is still quadratic in theory if a poorly designed hash function is used.
int copy_term(Term t, TermPtr des){
    deref(t);
    switch (tag(t)){
    case NUMVAR:
        *des = t;
        return 0;
    case ATM:
        *des = t;
        return atomic_hcode(t);
    case LST:
        p1 = untag(t);
        p2 = allocate_from_table(2);
        car_code = copy_term(*p1, p2);
        cdr_code = copy_term(*(p1+1), p2+1);
        hcode = seq_hcode(car_code, cdr_code);
        t1 = add_tag(p2, LST);
        if (is_ground_hcode(hcode)){
            t2 = hash_consing(t1, hcode);
            if (t1 != t2){
                deallocate_from_table(2);
                t1 = t2;
            }
        }
        *des = t1;
        return hcode;
    case STR:
        p1 = untag(t);
        sym = *p1;
        arity = get_arity(sym);
        p2 = allocate_from_table(arity+1);
        hcode = *p2 = sym;
        for (i=1;i<=arity;i++)
            hcode = seq_hcode(hcode, copy_term(*(p1+i), p2+i));
        t1 = add_tag(p2, STR);
        if (is_ground_hcode(hcode)){
            t2 = hash_consing(t1, hcode);
            if (t1 != t2){
                deallocate_from_table(arity+1);
                t1 = t2;
            }
        }
        *des = t1;
        return hcode;
    } /* end switch */
} /* end copy_term */

Fig. 2. Copy data with hash-consing.

time since the list \([1,\ldots,N]\) resides on the heap and for each descendant subgoal, the hash code of the argument is not available and hence has to be computed. To avoid this computation, we introduce input sharing.

### 5.2 Input Sharing

Input sharing amounts to letting a subgoal share its ground terms with its answers and descendant subgoals. Consider the tabled subgoal is_list([1,2,3]). The answer is the same as the subgoal, so it shares the term \([1,2,3]\) with the subgoal in the table area. The direct descendant subgoal is is_list([2,3]). Since the list \([2,3]\) is a suffix of \([1,2,3]\), the descendant subgoal should share it with the original subgoal in the table area.

To implement input sharing, we let the copying procedure set the frame slot of an argument of a tabled subgoal to the address of the copied argument in the
int copy_term(Term t, TermPtr des)
{
  deref(t);
  switch (tag(t))
  {
    case NUMVAR:
      *des = t;
      return 0;
    case ATM:
      *des = t;
      return atomic_hashcode(t);
    case LST:
      p1 = untag(t);
      if (!is_heap_reference(p1))
      {
        *des = t;
        return *(p1-1); /* return the tabled hash code */
      }
      p2 = allocate_from_table(3);
      car_code = copy_term(*p1, p2);
      cdr_code = copy_term(*(p1+1), p2+1);
      hcode = seq_hashcode(car_code, cdr_code);
      *(p2-1) = hcode;
      if (is_ground(hcode))
      {
        t1 = add_tag(p2, LST);
        if (t1 != t2)
        {
          deallocate_to_table(3);
          t1 = t2;
        }
      }
      *des = t1;
      return hcode;
    case STR:
      p1 = untag(t);
      if (!is_heap_reference(p1))
      {
        *des = t;
        return *(p1-1); /* return the tabled hash code */
      }
      sym = *p1;
      arity = get arity(sym);
      p2 = allocate_from_table(arity+2);
      p2++;
      hcode = *p2 = sym;
      for (i=1;i<=arity;i++)
      {
        hcode = seq_hashcode(hcode, copy_term(*(p1+i), p2+i));
      }
      *(p2-1) = hcode;
      if (is_ground(hcode))
      {
        t1 = add_tag(p2, STR);
        if (t1 != t2)
        {
          deallocate_to_table(arity+2);
          t1 = t2;
        }
      }
      *des = t1;
      return hcode;
    } /* end switch */
  } /* end switch */
}

Fig. 3. Tabling hash codes while copying with hash-consing.

table area if the argument is a ground structured term. So for the tabled subgoal is_list([1,2,3]), the frame slot of the argument initially references the list [1,2,3] on the heap. After the subgoal is copied to the table area, the frame slot is set to reference the copy of the list in the table area. In this way, the list will be shared by answers and the descendant subgoals. For programs that do not use de-


```c
int copy_subgoal_args(TermPtr src, TermPtr des, int arity)
{
    hcs = 0;
    for (i=1;i<arity;i++)
    {
        hcode = copy_term(*(src-i), des+i);
        if (is_a_ground(hcode)) *(src-i) = *(des+i);
        hcs = seq(hcs, hcode);
    }
    return hcs;
}
```

Fig. 4. Input sharing by updating frame slots.

Restructive assignments, which is the case for tabled programs, updating frame slots this way causes no problem.

The function `copy_subgoal_args` shown in Figure 4 implements input sharing. When an argument is found to be ground, the function lets the stack slot of the argument reference its copy in the table area. The function `copy_term` (in Figure 3) tests the reference to a compound term to see if the term needs to be copied. If it is not a heap reference, then the referenced term must reside in the table area and thus can be reused.

Note that our input sharing scheme has its limitation in the sense that it fails to facilitate sharing of ground components in non-ground arguments. Consider, for example, the subgoal `is_list([X,2,3])`. The suffix `[2,3]` will not be shared through input sharing in our implementation since the argument is not ground. It will eventually be shared through hash-consing, but its hash code needs to be computed again when it occurs in a descendant subgoal or an answer.

6 Evaluation

The improved tabling system described in this paper has been implemented and made available with B-Prolog version 7.7 (BP7.7). We evaluate the proposed approach by comparing BP7.7 with YAP (version 6.3.2) and XSB (version 3.3.6), and also the previous version of B-Prolog, version 7.6 (BP7.6), which did not have enhanced hash-consing. We also compare it with indexed programs produced by the transformation proposed in (Have and Christiansen 2012) running on B-Prolog 7.6 (indexed). We use the `is_list/1` predicate, the `edit_distance/3` program, and a PRISM program to show the effectiveness of the proposed techniques. We also test on a program that favors prefix sharing with tries more than suffix sharing with hash-consing. In addition, we also show results for the CHAT suite and the ATR parser, the traditional benchmarks used to evaluate tabling systems.

The results are obtained on a Linux machine with 16 2.4 GHz, 64 bit Intel Xeon(R) E7340 processor cores and 64 GB of memory. For this evaluation, only a single processor core is utilized. CPU times (in seconds) and table space (in kilobytes) consumptions are measured using the `statistics/1` built-in for BP and XSB, and `table_statistics/1` for YAP.

5 The source code is available in (Have and Christiansen 2012).
Table 1. Results on $\text{is\_list}([1,1,...,1])$

<table>
<thead>
<tr>
<th>N</th>
<th>BP7.7 time</th>
<th>BP7.7 space</th>
<th>BP7.6 time</th>
<th>BP7.6 space</th>
<th>indexed time</th>
<th>indexed space</th>
<th>YAP time</th>
<th>YAP space</th>
<th>XSB time</th>
<th>XSB space</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000</td>
<td>3.3</td>
<td>0.098</td>
<td>43</td>
<td>0.001</td>
<td>0.007</td>
<td>0.007</td>
<td>90</td>
<td>0.003</td>
<td>399</td>
</tr>
<tr>
<td>1000</td>
<td>0.001</td>
<td>66</td>
<td>0.776</td>
<td>86</td>
<td>0.003</td>
<td>78</td>
<td>0.033</td>
<td>180</td>
<td>0.040</td>
<td>567</td>
</tr>
<tr>
<td>1500</td>
<td>0.001</td>
<td>99</td>
<td>2.608</td>
<td>128</td>
<td>0.004</td>
<td>117</td>
<td>0.073</td>
<td>269</td>
<td>0.019</td>
<td>735</td>
</tr>
<tr>
<td>2000</td>
<td>0.002</td>
<td>131</td>
<td>6.169</td>
<td>171</td>
<td>0.005</td>
<td>156</td>
<td>0.134</td>
<td>359</td>
<td>0.037</td>
<td>903</td>
</tr>
<tr>
<td>2500</td>
<td>0.001</td>
<td>164</td>
<td>12.034</td>
<td>214</td>
<td>0.006</td>
<td>195</td>
<td>0.186</td>
<td>449</td>
<td>0.058</td>
<td>1071</td>
</tr>
<tr>
<td>3000</td>
<td>0.002</td>
<td>197</td>
<td>20.777</td>
<td>257</td>
<td>0.008</td>
<td>234</td>
<td>0.282</td>
<td>539</td>
<td>0.078</td>
<td>1239</td>
</tr>
<tr>
<td>3500</td>
<td>0.002</td>
<td>229</td>
<td>32.975</td>
<td>300</td>
<td>0.009</td>
<td>273</td>
<td>0.384</td>
<td>629</td>
<td>0.108</td>
<td>1407</td>
</tr>
<tr>
<td>4000</td>
<td>0.003</td>
<td>264</td>
<td>49.204</td>
<td>343</td>
<td>0.011</td>
<td>312</td>
<td>0.498</td>
<td>719</td>
<td>0.139</td>
<td>1575</td>
</tr>
<tr>
<td>4500</td>
<td>0.003</td>
<td>297</td>
<td>70.048</td>
<td>386</td>
<td>0.011</td>
<td>351</td>
<td>0.571</td>
<td>809</td>
<td>0.177</td>
<td>1743</td>
</tr>
<tr>
<td>5000</td>
<td>0.003</td>
<td>330</td>
<td>96.112</td>
<td>429</td>
<td>0.013</td>
<td>390</td>
<td>0.729</td>
<td>898</td>
<td>0.217</td>
<td>1911</td>
</tr>
</tbody>
</table>

Table 2. Results on $\text{is\_list}(L)$ where $L$ contains random data.

<table>
<thead>
<tr>
<th>N</th>
<th>BP7.7 time</th>
<th>BP7.6 time</th>
<th>BP7.6 space</th>
<th>indexed time</th>
<th>indexed space</th>
<th>YAP time</th>
<th>YAP space</th>
<th>XSB time</th>
<th>XSB space</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000</td>
<td>3.3</td>
<td>0.098</td>
<td>43</td>
<td>0.001</td>
<td>0.007</td>
<td>0.007</td>
<td>90</td>
<td>0.003</td>
</tr>
<tr>
<td>1000</td>
<td>0.001</td>
<td>66</td>
<td>0.776</td>
<td>86</td>
<td>0.003</td>
<td>78</td>
<td>0.033</td>
<td>180</td>
<td>0.040</td>
</tr>
<tr>
<td>1500</td>
<td>0.001</td>
<td>99</td>
<td>2.608</td>
<td>128</td>
<td>0.004</td>
<td>117</td>
<td>0.073</td>
<td>269</td>
<td>0.019</td>
</tr>
<tr>
<td>2000</td>
<td>0.002</td>
<td>131</td>
<td>6.169</td>
<td>171</td>
<td>0.005</td>
<td>156</td>
<td>0.134</td>
<td>359</td>
<td>0.037</td>
</tr>
<tr>
<td>2500</td>
<td>0.001</td>
<td>164</td>
<td>12.034</td>
<td>214</td>
<td>0.006</td>
<td>195</td>
<td>0.186</td>
<td>449</td>
<td>0.058</td>
</tr>
<tr>
<td>3000</td>
<td>0.002</td>
<td>197</td>
<td>20.777</td>
<td>257</td>
<td>0.008</td>
<td>234</td>
<td>0.282</td>
<td>539</td>
<td>0.078</td>
</tr>
<tr>
<td>3500</td>
<td>0.002</td>
<td>229</td>
<td>32.975</td>
<td>300</td>
<td>0.009</td>
<td>273</td>
<td>0.384</td>
<td>629</td>
<td>0.108</td>
</tr>
<tr>
<td>4000</td>
<td>0.003</td>
<td>264</td>
<td>49.204</td>
<td>343</td>
<td>0.011</td>
<td>312</td>
<td>0.498</td>
<td>719</td>
<td>0.139</td>
</tr>
<tr>
<td>4500</td>
<td>0.003</td>
<td>297</td>
<td>70.048</td>
<td>386</td>
<td>0.011</td>
<td>351</td>
<td>0.571</td>
<td>809</td>
<td>0.177</td>
</tr>
<tr>
<td>5000</td>
<td>0.003</td>
<td>330</td>
<td>96.112</td>
<td>429</td>
<td>0.013</td>
<td>390</td>
<td>0.729</td>
<td>898</td>
<td>0.217</td>
</tr>
</tbody>
</table>

Table 1 shows the results on the query $\text{is\_list}([1,1,...,1])$ where $N$ is the number of 1s in the list. All the systems except for BP7.6 demonstrate a close-to-linear complexity. The higher time complexity of BP7.6 is due to the fact that BP7.6 only uses the first three elements of a list as the key and hashing degenerates into linear search for the query because of hash collision. The difference in time among BP7.7, YAP and XSB is at least a large constant factor. As mentioned above, a trie allows for sharing of prefixes while hash-consing allows for sharing of suffixes as long as lists are concerned. For a list that contains repeated data, there are an equal number of prefixes and suffixes, and hence both types of sharing are equally favored. The difference between BP7.7 and indexed is only a small constant factor.

Table 2 shows the results on the query $\text{is\_list}(L)$ where $L$ is a list of random constants. BP consumes linear space and linear time; YAP consumes linear space thanks to the global trie for terms but takes quadratic time; XSB is quadratic in both time and space. For random lists, suffix sharing with hash consing is clearly more effective than prefix sharing with tries.

Tables 3 and 4 show the results on the edit_distance program with repeated

---

6 A random number generator is used to generate the lists. For each size, the same list was used for all the systems.
Table 3. Results on edit([1,1,...,1],[1,1,...,1],D).

<table>
<thead>
<tr>
<th>N</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.005</td>
<td>60</td>
<td>0.026</td>
<td>97</td>
<td>0.003</td>
<td>90</td>
<td>0.005</td>
<td>213</td>
<td>0.006</td>
<td>1273</td>
</tr>
<tr>
<td>60</td>
<td>0.003</td>
<td>233</td>
<td>0.726</td>
<td>378</td>
<td>0.016</td>
<td>348</td>
<td>0.034</td>
<td>819</td>
<td>0.057</td>
<td>4341</td>
</tr>
<tr>
<td>90</td>
<td>0.007</td>
<td>519</td>
<td>5.189</td>
<td>841</td>
<td>0.036</td>
<td>776</td>
<td>0.107</td>
<td>1820</td>
<td>0.235</td>
<td>9435</td>
</tr>
<tr>
<td>120</td>
<td>0.015</td>
<td>917</td>
<td>21.216</td>
<td>1487</td>
<td>0.064</td>
<td>1372</td>
<td>0.266</td>
<td>3214</td>
<td>0.736</td>
<td>16554</td>
</tr>
<tr>
<td>150</td>
<td>0.022</td>
<td>1427</td>
<td>65.596</td>
<td>2316</td>
<td>0.102</td>
<td>2137</td>
<td>0.517</td>
<td>8222</td>
<td>1.635</td>
<td>25698</td>
</tr>
<tr>
<td>180</td>
<td>0.031</td>
<td>2051</td>
<td>156.072</td>
<td>3328</td>
<td>0.142</td>
<td>3071</td>
<td>0.942</td>
<td>7183</td>
<td>3.041</td>
<td>36868</td>
</tr>
<tr>
<td>210</td>
<td>0.047</td>
<td>2786</td>
<td>334.190</td>
<td>4523</td>
<td>0.208</td>
<td>4173</td>
<td>1.533</td>
<td>9759</td>
<td>5.035</td>
<td>50064</td>
</tr>
<tr>
<td>240</td>
<td>0.060</td>
<td>3634</td>
<td>646.550</td>
<td>5900</td>
<td>0.267</td>
<td>5445</td>
<td>2.367</td>
<td>12728</td>
<td>7.662</td>
<td>65285</td>
</tr>
<tr>
<td>270</td>
<td>0.074</td>
<td>4595</td>
<td>1159.182</td>
<td>7460</td>
<td>0.339</td>
<td>6885</td>
<td>3.081</td>
<td>16090</td>
<td>11.327</td>
<td>82531</td>
</tr>
<tr>
<td>300</td>
<td>0.095</td>
<td>5668</td>
<td>1955.331</td>
<td>9204</td>
<td>0.448</td>
<td>8493</td>
<td>4.401</td>
<td>19847</td>
<td>15.664</td>
<td>101803</td>
</tr>
</tbody>
</table>

Table 4. Results on edit(L1,L2,D) where L1 and L2 contain random data.

<table>
<thead>
<tr>
<th>N</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.001</td>
<td>61</td>
<td>0.000</td>
<td>97</td>
<td>0.004</td>
<td>90</td>
<td>0.006</td>
<td>214</td>
<td>0.011</td>
<td>4148</td>
</tr>
<tr>
<td>60</td>
<td>0.003</td>
<td>234</td>
<td>0.006</td>
<td>378</td>
<td>0.020</td>
<td>348</td>
<td>0.045</td>
<td>822</td>
<td>0.099</td>
<td>2706</td>
</tr>
<tr>
<td>90</td>
<td>0.010</td>
<td>521</td>
<td>0.016</td>
<td>841</td>
<td>0.038</td>
<td>776</td>
<td>0.118</td>
<td>1823</td>
<td>0.313</td>
<td>89645</td>
</tr>
<tr>
<td>120</td>
<td>0.017</td>
<td>919</td>
<td>0.033</td>
<td>1487</td>
<td>0.067</td>
<td>1372</td>
<td>0.298</td>
<td>3218</td>
<td>0.759</td>
<td>290183</td>
</tr>
<tr>
<td>150</td>
<td>0.027</td>
<td>1430</td>
<td>0.057</td>
<td>2316</td>
<td>0.105</td>
<td>2137</td>
<td>0.591</td>
<td>5007</td>
<td>1.501</td>
<td>404752</td>
</tr>
<tr>
<td>180</td>
<td>0.038</td>
<td>2054</td>
<td>0.094</td>
<td>3328</td>
<td>0.148</td>
<td>3071</td>
<td>1.058</td>
<td>7190</td>
<td>2.771</td>
<td>695363</td>
</tr>
<tr>
<td>210</td>
<td>0.056</td>
<td>2790</td>
<td>0.156</td>
<td>4523</td>
<td>0.217</td>
<td>4173</td>
<td>1.695</td>
<td>9766</td>
<td>4.271</td>
<td>1099906</td>
</tr>
<tr>
<td>240</td>
<td>0.073</td>
<td>3639</td>
<td>0.219</td>
<td>5900</td>
<td>0.282</td>
<td>5445</td>
<td>2.687</td>
<td>12736</td>
<td>6.247</td>
<td>1637354</td>
</tr>
<tr>
<td>270</td>
<td>0.092</td>
<td>4600</td>
<td>0.297</td>
<td>7460</td>
<td>0.352</td>
<td>6885</td>
<td>3.782</td>
<td>16100</td>
<td>8.787</td>
<td>2392726</td>
</tr>
<tr>
<td>300</td>
<td>0.114</td>
<td>5674</td>
<td>0.435</td>
<td>9204</td>
<td>0.466</td>
<td>8493</td>
<td>5.248</td>
<td>19857</td>
<td>11.954</td>
<td>3187340</td>
</tr>
</tbody>
</table>

data and random data, respectively. The main predicate edit(L1,L2,D) in the program computes the distance between L1 and L2, i.e., the number of substitutions, insertions and deletions needed to transform L1 to L2. The tabled version finds all solutions. BP7.7 is significantly faster than BP7.6 on the type of queries that use repeated data. BP7.7 also outperforms YAP and XSB in both time and space on both types of queries. Similar to the is_list benchmark, enhanced hash-consing is asymptotically more effective than tries on random data.

Table 5 compares BP7.7 and BP7.6 on the PRISM program that simulates a two-state hidden Markov model (Sato et al. 2010). For our benchmarking purpose, the training data of the form hmm([a,b,a,b,...]) are used, and only the time and space required to find all the explanations are measured. While BP7.7 consumes slightly more space than BP7.6 due to the overhead of hash-consing, it outperforms BP7.6 in time by a linear factor.

Although it is more common for subgoals of recursive programs to share suffixes than prefixes, it is possible to find programs on which prefix sharing with tries is more effective than suffix sharing with hash-consing. The following gives such a program:

```prolog
:-table create_list/2.
create_list(N,L):-
```
Table 5. Results on the PRISM program HMM.

<table>
<thead>
<tr>
<th>N</th>
<th>time</th>
<th>space</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0.002</td>
<td>222</td>
<td>1.164</td>
<td>179</td>
</tr>
<tr>
<td>3000</td>
<td>0.005</td>
<td>333</td>
<td>3.911</td>
<td>269</td>
</tr>
<tr>
<td>4000</td>
<td>0.006</td>
<td>444</td>
<td>9.249</td>
<td>399</td>
</tr>
<tr>
<td>5000</td>
<td>0.008</td>
<td>555</td>
<td>18.044</td>
<td>449</td>
</tr>
<tr>
<td>6000</td>
<td>0.010</td>
<td>666</td>
<td>31.150</td>
<td>539</td>
</tr>
<tr>
<td>7000</td>
<td>0.011</td>
<td>776</td>
<td>49.441</td>
<td>628</td>
</tr>
<tr>
<td>8000</td>
<td>0.013</td>
<td>889</td>
<td>73.774</td>
<td>718</td>
</tr>
<tr>
<td>9000</td>
<td>0.015</td>
<td>1000</td>
<td>105.049</td>
<td>808</td>
</tr>
<tr>
<td>10000</td>
<td>0.018</td>
<td>1111</td>
<td>144.140</td>
<td>898</td>
</tr>
</tbody>
</table>

Table 6. Results on create_list(N,L).

<table>
<thead>
<tr>
<th>N</th>
<th>BP7.7</th>
<th>BP7.6</th>
<th>YAP</th>
<th>XSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>space</td>
<td>time</td>
<td>space</td>
</tr>
<tr>
<td>500</td>
<td>0.035</td>
<td>2417</td>
<td>0.107</td>
<td>990</td>
</tr>
<tr>
<td>1000</td>
<td>0.201</td>
<td>9564</td>
<td>0.827</td>
<td>3937</td>
</tr>
<tr>
<td>1500</td>
<td>0.654</td>
<td>21635</td>
<td>2.989</td>
<td>8831</td>
</tr>
<tr>
<td>2000</td>
<td>0.969</td>
<td>37926</td>
<td>7.245</td>
<td>15679</td>
</tr>
<tr>
<td>2500</td>
<td>2.151</td>
<td>60082</td>
<td>14.130</td>
<td>24480</td>
</tr>
<tr>
<td>3000</td>
<td>2.660</td>
<td>85890</td>
<td>24.343</td>
<td>35249</td>
</tr>
<tr>
<td>3500</td>
<td>3.276</td>
<td>116011</td>
<td>38.397</td>
<td>47956</td>
</tr>
<tr>
<td>4000</td>
<td>4.011</td>
<td>150192</td>
<td>57.217</td>
<td>62616</td>
</tr>
<tr>
<td>4500</td>
<td>7.319</td>
<td>194310</td>
<td>80.994</td>
<td>79229</td>
</tr>
<tr>
<td>5000</td>
<td>8.316</td>
<td>238885</td>
<td>110.631</td>
<td>97796</td>
</tr>
</tbody>
</table>

between(1,N,I),
range(1,I,L).

The query create_list(N,L) creates N lists [1], [1,2], ..., and [1,2,...,N] that have only common prefixes. As shown in Table 6, XSB consumes linear space, while BP and YAP consume quadratic space. YAP tables all suffixes into the global trie for terms and there are \(O(n^2)\) suffixes. BP7.7 consumes more table space than BP7.6 since all the terms are hash-consed but none is shared. BP7.6 is slower than BP7.7 since the hash function used in BP7.6, which is based on the first three elements of a list, results in more collisions than BP7.7.

Table 7 compares the systems on the CHAT benchmark suite and the ATR parser. There is almost no difference between BP7.7 and BP7.6 in time and the space overhead incurred by hash-consing is noticeable. Hash-consing has no positive effect on these programs because the sequences used in the programs are very short.

7 Related Work

Since structure sharing (Boyer and Moore 1972) was discarded and the Warren Abstract Machine (WAM) (Warren 1983) triumphed as the implementation model of Prolog, there has been little attention paid to exploiting data sharing in Prolog.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BP 7.7 time</th>
<th>BP 7.6 space</th>
<th>YAP time</th>
<th>YAP space</th>
<th>XSB time</th>
<th>XSB space</th>
</tr>
</thead>
<tbody>
<tr>
<td>cso</td>
<td>0.015</td>
<td>198</td>
<td>0.0129</td>
<td>11</td>
<td>0.009</td>
<td>26</td>
</tr>
<tr>
<td>cso</td>
<td>0.025</td>
<td>332</td>
<td>0.026</td>
<td>11</td>
<td>0.019</td>
<td>27</td>
</tr>
<tr>
<td>dieg</td>
<td>0.008</td>
<td>108</td>
<td>0.009</td>
<td>11</td>
<td>0.006</td>
<td>23</td>
</tr>
<tr>
<td>gabriel</td>
<td>0.011</td>
<td>111</td>
<td>0.012</td>
<td>9</td>
<td>0.006</td>
<td>20</td>
</tr>
<tr>
<td>kalah</td>
<td>0.008</td>
<td>90</td>
<td>0.008</td>
<td>15</td>
<td>0.006</td>
<td>35</td>
</tr>
<tr>
<td>pg</td>
<td>0.006</td>
<td>69</td>
<td>0.006</td>
<td>7</td>
<td>0.004</td>
<td>15</td>
</tr>
<tr>
<td>read</td>
<td>0.057</td>
<td>987</td>
<td>0.058</td>
<td>23</td>
<td>0.099</td>
<td>46</td>
</tr>
<tr>
<td>atr</td>
<td>0.509</td>
<td>15111</td>
<td>0.543</td>
<td>5947</td>
<td>0.325</td>
<td>52520</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.280</td>
<td>45400</td>
</tr>
</tbody>
</table>

In his Diploma thesis (Neumerkel 1989), Ulrich Neumerkel gave several example Prolog programs that would consume an order-of-magnitude less space with data sharing than without sharing. He proposed applying hash-consing and DFA-minimization to sharing terms including cyclic ones. The proposed approach would incur considerable overhead if every compound term is hash-consed when created, and hence it is infeasible to incorporate the approach into the WAM. Following Appel and Goncalves’s hash-consing garbage collector for SML/NJ (Appel and de Rezende Goncalves 2003), Nguyen and Demoen recently built a similar garbage collector for hProlog (Nguyen and Demoen 2012). The garbage collector hash-conses compound terms on the heap in one phase and performs absorption in another phase such that for the replications of a compound term only one copy is kept and all the others are garbage collected. Their experiment basically confirms the disappointing result reported in Appel and Goncalves’s paper: the overhead outweighs the gain except for special programs.

Hash-consing can be applied to the built-in predicate `findall/3`, as suggested by O’Keefe (O’Keefe 2001), to avoid repeatedly copying the same term in different answers. Currently, B-Prolog is the only Prolog system that supports hash-consing for `findall/3`. It employs a hash table for ground terms in the `findall` area. The algorithm and memory manager developed for the table area is reused for the `findall` area. With hash-consing, the system copies a ground term only once when copying answers from the `findall` area to the heap. Input sharing is exploited in the same way as for tabled subgoals. For a `findall` call, the compiler converts it into a call to a temporary predicate such that each argument of the generator occupies one slot in the stack frame. At runtime, the system first copies the arguments of the generator from the stack/heap to the `findall` area before the generator is executed. When an argument of the generator is found to be a ground compound term, its frame slot is set to reference the copy in the `findall` area. In this way, the argument and its subterms can be reused by the answers and the descendant calls. Nguyen and Demoen’s implementation of input sharing for `findall/3` (Nguyen and Demoen 2012) distinguishes between old terms that are created before the generator and new terms that are generated by the generator, and have answers share the old

---

7 A lot of work has been done on indexing Prolog terms, but indexing is a different kind of sharing since it does not consider reuse of terms from different sources.
terms. Their scheme can exploit sharing of not only ground arguments but also ground terms in non-ground arguments. Their scheme may not be suited for tabled data since, unlike data in the findall area which live and die with the generator, tabled data are permanent. Also, their implementation does not exploit output sharing.

A trie has been a popular data structure for organizing tabled subgoals and answers (Ramakrishnan et al. 1998). It is adopted by all the tabled Prolog systems except B-Prolog. As far as lists are concerned, a trie facilitates sharing of the prefixes while hash-consing allows for sharing of the suffixes. So for the two lists \([1,2]\) and \([1,2,3]\), the former shares the same path as the latter in the trie, but they are treated as separate lists when hash-consed; for the two lists \([2,3]\) and \([1,2,3]\), however, a trie allows for no sharing while hash-consing allows for complete sharing.

Another advantage of tries is that they can be used to perform both variant testing and subsumption testing, and thus can be used in both variant-based and subsumption-based tabling systems. Hash-consing, on the other hand, can be used to perform equivalence testing only and thus cannot directly be used for subsumption-based tabling.

Terms stored in a trie have a different representation from terms on the heap. For example, in the YAP system, tries are represented as trie instructions (Santos Costa et al. 2012). For this reason, when an answer is returned, it must be copied from its trie in the table area to the heap even if it is ground. In our system, structured ground terms in the table area have exactly the same representation as on the heap, so when they occur in an answer they do not need to be copied when the answer is returned.

In the original implementation of XSB and YAP, one trie is used for all tabled subgoals, and for each subgoal one trie is used for the answer table. To enhance sharing, Raimundo and Rocha propose using a global trie for all tabled structured terms (Raimundo and Rocha 2011). Due to the necessity of copying answers from the table area to the heap, the time complexity remains the same even when the space complexity drops.

To some extent, the idea of representing sentences as position indexed facts (Have and Christiansen 2012; Swift et al. 2009) is similar to hash-consing in the sense that a hash-consed term always is associated with a hash code. The translation from a program that deals with sequences represented as lists into one that uses position representation is not trivial. When difference lists are involved, the translation is even more complicated. The program obtained after translation may lose sharing opportunities. Therefore, hash-consing is a more practical solution to sharing than program transformation.

As far as we know, our implementation is the first attempt to apply hash-consing to tabling. Our implementation enhances hash-consing with input sharing and hash code memoization to speed-up computation of hash codes. The extra cell used to store the hash code of a compound term is overhead if the term is never shared. Nevertheless, while the increase of space is always a constant factor, the gain in speed can be linear in the size of the data.
8 Conclusion

We have presented an implementation of hash-consing for tabling structured data. Hash-consing facilitates sharing of structured data and can eliminate the extra linear factor of space complexity commonly seen in early tabling systems when dealing with sequences. Hash-consing alone does not change the time complexity. We have enhanced it with input sharing and hash code memoization to eliminate the extra linear factor of time complexity in dealing with sequences. The resulting tabling system significantly improves the scalability of language parsing and bio-sequence analysis applications.

Our work will shed some light on the discussion on what data structure to use for tabled data. A trie is suitable for sharing prefixes and hash-consing is suitable for sharing suffixes of sequences. Although it is possible to find programs that make prefix sharing arbitrarily better than suffix sharing, it is more common for subgoals of recursive programs to share suffixes than prefixes. Therefore, hash-consing is in general a better choice than tries as a data structure for representing tabled data. Hash-consing as it is in our implementation is not suitable for subsumption-based tabling. It is future work to adapt hash-consing to subsumption testing.

Acknowledgements

The PRISM system has been the motivation for this project and we thank Taisuke Sato and Yoshitaka Kameya for their discussion. We also thank the anonymous referees for their detailed comments on the presentation. Neng-Fa Zhou was supported in part by NSF (No.1018006) and Christian Theil Have was supported by the project Logic-statistic modelling and analysis of biological sequence data funded by the NABIIT program under the Danish Strategic Research Council.

References

Efficient Tabling of Structured Data with Enhanced Hash-Consing


