Efficient tabling of structured data with enhanced hash-consing

Zhou, Neng-Fa; Have, Christian Theil

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

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

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

```
:-table is_list/1.
is_list([]).
is_list([_|L]):-is_list(L).
```

For the subgoal `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 `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 `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 `'C'/3`.[1] 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. *Hash-consing*

---

[1] Personal communication with David S. Warren, 2011.

(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 `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 `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 *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]

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 *pioneer* or *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:

```
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.

```
int copy_subgoal_args(TermPtr src, TermPtr des, int arity){
   hcsum = 0;
   for (i=1;i<=arity;i++){
       hcode = copy_term(*(src-i), des+i);
       hc_sum = seq_hcode(hc_sum,hcode);
   }
   return hc_sum;
}

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);
       *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);
       *des = t1;
       return hcode;
   } /* end switch */
} /* end copy_term */
```

Fig. 1. Copy data to the table area with no sharing.

The function `copy_term` exploits no sharing of data. Consider, for example, the following program and the query `is_list([1,2])`. After completion of the query, the subgoal table contains three tabled subgoals, `is_list([1,2])`, `is_list([2])`, and `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 `[1,2]` and two separate copies of `[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 `is_list([1,2,...,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

before (Goto 1974). 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_to_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_to_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,...,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_hcode(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);
        p2++;
        car_code = copy_term(*p1, p2);
        cdr_code = copy_term(*(p1+1), p2+1);
        hcode = seq_hcode(car_code,cdr_code);
        *(p2-1) = hcode;
        t1 = add_tag(p2,LST);
        if (is_ground_hcode(hcode)){
            t2 = hash_consing(t1,hcode);
            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_hcode(hcode, copy_term(*(p1+i), p2+i));
        *(p2-1) = hcode;
        t1 = add_tag(p2,STR);
        if (is_ground_hcode(hcode)){
            t2 = hash_consing(t1,hcode);
            if (t1 != t2){
                deallocate_to_table(arity+2);
                t1 = t2;
            }
        }
        *des = t1;
        return hcode;
    } /* end switch */
} /* end copy_term */
```

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

table area if the argument is a ground structured term. So for the tabled sub-goal 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-

```
int copy_subgoal_args(TermPtr src, TermPtr des, int arity){
   hcsum = 0;
   for (i=1;i<=arity;i++){
      hcode = copy_term(*(src-i), des+i);
      if (is_ground_hcode(hcode)) *(src-i) = *(des+1);
      hc_sum = seq_hcode(hc_sum,hcode);
   }
   return hc_sum;
}
```

Fig. 4. Input sharing by updating frame slots.

structive 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`[5] 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* `is_list([1,1,...,1])`

N	BP7.7		BP7.6		*indexed*		YAP		XSB	
	time	space	time	space	time	space	time	space	time	space
500	0.000	33	0.098	43	0.001	39	0.007	90	0.003	399
1000	0.001	66	0.776	86	0.003	78	0.033	180	0.010	567
1500	0.001	99	2.608	128	0.004	117	0.073	269	0.019	735
2000	0.002	131	6.169	171	0.005	156	0.134	359	0.037	903
2500	0.001	164	12.034	214	0.006	195	0.186	449	0.058	1071
3000	0.002	197	20.777	257	0.008	234	0.282	539	0.078	1239
3500	0.002	229	32.975	300	0.009	273	0.384	629	0.108	1407
4000	0.003	264	49.204	343	0.011	312	0.498	719	0.139	1575
4500	0.003	297	70.048	386	0.011	351	0.571	809	0.177	1743
5000	0.003	330	96.112	429	0.013	390	0.729	898	0.217	1911

Table 2. *Results on* `is_list(L)` *where* L *contains random data.*

N	BP7.7		BP7.6		*indexed*		YAP		XSB	
	time	space	time	space	time	space	time	space	time	space
500	0.000	33	0.000	43	0.002	39	0.008	90	0.024	9990
1000	0.001	66	0.001	86	0.002	78	0.032	180	0.063	39236
1500	0.001	99	0.001	128	0.004	117	0.082	270	0.142	87991
2000	0.001	132	0.002	171	0.005	156	0.134	360	0.252	156269
2500	0.001	164	0.003	214	0.007	195	0.218	450	0.387	244071
3000	0.002	197	0.003	257	0.008	234	0.341	540	0.559	351401
3500	0.002	229	0.004	300	0.010	273	0.401	630	0.766	478260
4000	0.003	264	0.005	343	0.011	312	0.537	719	0.978	624640
4500	0.003	297	0.006	386	0.012	351	0.703	809	1.244	790555
5000	0.004	330	0.008	429	0.013	390	0.894	899	1.504	975990

Table 1 shows the results on the query `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 that 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 `is_list(L)` where L is a list of random constants.[6] 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)`.

N	BP7.7		BP7.6		*indexed*		YAP		XSB	
	time	space	time	space	time	space	time	space	time	space
30	0.000	60	0.026	97	0.003	90	0.005	213	0.006	1273
60	0.003	233	0.726	378	0.016	348	0.034	819	0.057	4341
90	0.007	519	5.189	841	0.036	776	0.107	1820	0.235	9435
120	0.015	917	21.216	1487	0.064	1372	0.266	3214	0.736	16554
150	0.022	1427	63.536	2316	0.102	2137	0.517	5002	1.635	25698
180	0.031	2051	156.072	3328	0.142	3071	0.942	7183	3.041	36868
210	0.047	2786	334.190	4523	0.208	4173	1.533	9759	5.035	50064
240	0.060	3634	646.550	5900	0.267	5445	2.367	12728	7.662	65285
270	0.074	4595	1159.182	7460	0.339	6885	3.081	16090	11.327	82531
300	0.095	5668	1955.331	9204	0.448	8493	4.401	19847	15.664	101803

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

N	BP7.7		BP7.6		*indexed*		YAP		XSB	
	time	space	time	space	time	space	time	space	time	space
30	0.001	61	0.000	97	0.004	90	0.005	214	0.011	4148
60	0.003	234	0.006	378	0.020	348	0.045	822	0.099	27706
90	0.010	521	0.016	841	0.038	776	0.118	1823	0.313	89645
120	0.017	919	0.033	1487	0.067	1372	0.298	3218	0.759	209183
150	0.027	1430	0.057	2316	0.105	2137	0.591	5007	1.501	404752
180	0.038	2054	0.094	3328	0.148	3071	1.058	7190	2.771	695363
210	0.056	2790	0.156	4523	0.217	4173	1.695	9766	4.271	1099906
240	0.073	3639	0.219	5900	0.282	5445	2.687	12736	6.247	1637354
270	0.092	4600	0.297	7460	0.352	6885	3.782	16100	8.787	2327276
300	0.114	5674	0.435	9204	0.466	8493	5.248	19857	11.954	3187340

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:

```
:-table create_list/2.
create_list(N,L):-
```

Table 5. *Results on the PRISM program HMM.*

	BP7.7		BP7.6	
N	time	space	time	space
2000	0.002	222	1.164	179
3000	0.005	333	3.911	269
4000	0.006	444	9.249	359
5000	0.008	555	18.044	449
6000	0.010	666	31.150	539
7000	0.011	776	49.441	628
8000	0.013	889	73.774	718
9000	0.015	1000	105.049	808
10000	0.018	1111	144.140	898

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

	BP7.7		BP7.6		YAP		XSB	
N	time	space	time	space	time	space	time	space
500	0.035	2417	0.107	990	0.039	3965	0.023	290
1000	0.201	9564	0.827	3937	0.201	15742	0.043	348
1500	0.654	21635	2.989	8831	0.523	35332	0.095	407
2000	0.969	37926	7.245	15679	0.962	62734	0.169	465
2500	2.151	60082	14.130	24480	1.699	97949	0.264	524
3000	2.660	85890	24.343	35249	2.630	140976	0.378	583
3500	3.276	116011	38.397	47956	3.739	191816	0.517	641
4000	4.011	150192	57.217	62616	5.071	250468	0.675	700
4500	7.319	194310	80.994	79229	6.978	316933	0.853	758
5000	8.316	238885	110.631	97796	9.267	391211	1.051	817

```
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 $O(\text{N}^2)$ 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 7. *Results on the CHAT benchmarks and the ATR parser.*

Benchmark	BP 7.7		BP 7.6		YAP		XSB	
	time	space	time	space	time	space	time	space
cs_o	0.015	198	0.0129	11	0.009	26	0.011	285
cs_r	0.025	332	0.026	11	0.019	27	0.022	286
disj	0.008	108	0.009	11	0.005	23	0.007	277
gabriel	0.011	111	0.012	9	0.006	20	0.008	272
kalah	0.008	90	0.008	15	0.006	35	0.008	304
pg	0.006	69	0.006	7	0.004	15	0.006	263
read	0.057	987	0.058	23	0.099	46	0.030	327
atr	0.509	15111	0.543	5947	0.325	52520	0.280	45400

implementations.[7] 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

References

Appel, A. W. and de Rezende Goncalves, M. J. 2003. Hash-consing garbage collection. Technical Report TR 74-03, Princeton University.

Boyer, R. S. and Moore, J. S. 1972. A sharing of structure in theorem proving programs. *Machine Intelligence 7*, 101–116.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1, 20–74.

Ershov, A. 1959. On programming of arithmetic operations. *Communications of the ACM 1,* 8, 3–6.

Goto, E. 1974. Monocopy and associative algorithms in extended Lisp. Technical Report TR 74-03, University of Tokyo.

Have, C. T. and Christiansen, H. 2012. Efficient tabling of structured data using indexing and program transformation. In *PADL*. LNCS 7149, 93–107.

Michie, D. 1968. "memo" functions and machine learning. *Nature*, 19–22.

Neumerkel, U. 1989. Garbage collection in Prolog systems (in German). Ph.D. thesis, Thesis, Technical University of Vienna.

Nguyen, P.-L. and Demoen, B. 2012. Representation sharing for Prolog. *TPLP*.

O'Keefe, R. A. 2001. O(1) reversible tree navigation without cycle. *TPLP 1,* 5, 617–630.

Raimundo, J. and Rocha, R. 2011. Global trie for subterms. In *CICLOPS*.

Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., and Warren, D. 1998. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming 38*, 31–54.

Santos Costa, V., Rocha, R., and Damas, L. 2012. The YAP Prolog system. *TPLP, Special Issue on Prolog Systems 12,* 1-2, 5–34.

Sato, T. and Kameya, Y. 2008. New advances in logic-based probabilistic modeling by PRISM. In *Probabilistic Inductive Logic Programming*. 118–155.

Sato, T., Zhou, N.-F., Kameya, Y., and Yizumi, Y. 2010. The PRISM user's manual. http://www.mi.cs.titech.ac.jp/prism/.

Somogyi, Z. and Sagonas, K. 2006. Tabling in Mercury: Design and implementation. In *PADL*. LNCS 3819, 150–167.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with tabled logic programming. *TPLP, Special issue on Prolog systems 12,* 1-2, 157–187.

Swift, T., Warren, D. S., et al. 2009. The XSB Programmer's Manual: vols. 1 and 2. http://xsb.sf.net.

Warren, D. H. D. 1983. An abstract Prolog instruction set. Technical note 309, SRI International.

Zhou, N.-F. 2012. The language features and architecture of B-Prolog. *TPLP, Special Issue on Prolog Systems 12,* 1-2, 189–218.

Zhou, N.-F., Sato, T., and Shen, Y.-D. 2008. Linear tabling strategies and optimizations. *TPLP 8,* 1, 81–109.