Efficient tabling of structured data using indexing and program transformation

Have, Christian Theil; Christiansen, Henning

Published in:
Lecture Notes in Computer Science

DOI:
10.1007/978-3-642-27694-1

Publication date:
2012

Document Version
Early version, also known as pre-print

Citation for published version (APA):
Have, C. T., & Christiansen, H. (2012). Efficient tabling of structured data using indexing and program transformation. *Lecture Notes in Computer Science*, *7149*, 93-107. https://doi.org/10.1007/978-3-642-27694-1

Efficient Tabling of Structured Data Using Indexing and Program Transformation

Christian Theil Have and Henning Christiansen

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: {cth, henning}@ruc.dk

Abstract. Tabling of structured data is important to support dynamic programming in logic programs. Several existing tabling systems for Prolog do not efficiently deal with structured data, but duplicate part of the structured data in different instances of tabled goals. As a consequence, time and space complexity may often be significantly higher than the theoretically optimal. A simple program transformation is proposed which uses an indexing of structured data that eliminates this problem, and drastic improvements of time and space complexity can be demonstrated. The technique is demonstrated for dynamic programming examples expressed in Prolog and in PRISM.

1 Introduction

Tabling in logic programming systems is an established technique which can give a significant speed-up of program execution and make it easier to write efficient programs in a declarative style. Basically, tabling means that the system maintains a table of calls and their answers and each time a new call is entered, it is checked if it (or a perhaps more general call, cf. [15]) is stored in the table already; if so, there is no need to execute it again and a previously found solution is used. It is included in several recognized Prolog systems such as B-Prolog [17], XSB [13] and YAP [7].

However, we can demonstrate that these systems may waste unnecessary time and space for copying and matching structures in situations where operations on single pointers could have been used instead. This can be the case when a program is called with a huge, ground structure as one of its arguments, and this argument is decomposed into sub-structures which are tabled independently.

In addition to pointing out the problem, we can show how it can be bypassed by a straightforward program transformation and a few auxiliary predicates that can be written in plain Prolog. A significant speed-up is demonstrated for selected test programs. In a longer perspective, we advocate such techniques be incorporated into logic programming systems with tabling such as those mentioned, where it can be implemented at a lower level where machine address pointers are available rather that using a high-level “simulation” of pointers as we do here.

Our own background for working with this problem is work on analysis of biological sequence data using the probabilistic-logic system PRISM [8] which is implemented on top of B-Prolog and which is heavily dependent on its tabling mechanism. Together with another general program transformation based technique that we have developed [2], which improves the performance of tabling in the presence of non-discriminatory arguments, the technique described in the present paper increases significantly the size of sequences that can be meaningfully analyzed by means of PRISM programs.

Section 2 introduces the problem with tabling of structured data through an example. Section 3 describes an indexed representation of structured data that circumvents the problem, and section 4 demonstrates the effect for two problems, a dynamic programming problem in Prolog in section 4.1 and a PRISM program in section 4.2. Section 5 describes a general and automatic program transformation. Section 6 discuss limitations of our approach. Section 7 describes related work and section 8 sums up and discuss future work.

2 The Trouble with Tabling Structured Data

In this section we empirically demonstrate that all major Prolog tabling systems have a problem with structured data. Through the benchmarking of an implementation of the `last/2` predicate — which traverses a list to find the last element — we observe that when this predicate is tabled, time and space complexity is far worse than without tabling.

The following is a straight-forward implementation of the `last/2` predicate.

```
last([X],X).
last([_|L],X) :-
    last(L,X).
```

If `last/2` is called with a list `L` of length N, e.g. `last(L,_)`, then the expected time-complexity of this implementation is clearly $O(N)$. However, if the predicate is tabled, then the tabling system may have to store N partial copies of the list, e.g. the first copy will be the full list, the second copy will just store $N-1$ elements, and so on until every possible tail down to the last element of the list has been tabled. This results in $O(N^2)$ tabled list elements.

Naive copying of the lists hence make the tabled version of `last/2` (at least) quadratic — with regard to both time and space consumption — rather than linear as in the non-tabled version. Tabling systems do employ some advanced techniques to avoid the expensive copying and which may reduce memory consumption and/or time complexity. For instance, B-Prolog uses hashing of goals [18], XSB uses a trie data structure [13] and Yap [7] uses a trie structure, which in [6] is refined into a so-called global trie which applies a sharing strategy for common subterms whenever possible. This can reduce space consumption, but since there is no sharing between the trie and the actual arguments of an active call, each execution of a call may typically involve a full traversal of its arguments.

Nevertheless, as can be witnessed from Figure 1, all tabling systems pay a price for structured data in either time or space. The figure shows time and space consumption for `last(L,_)` with varying sizes of `L`, where `L` is either a list of consecutive ones or a list of random numbers generated using the following simple random number generator.

```
random_list(0,_,[]).
random_list(N,Prev,[X|L]):-
    B is (9381*Prev + 12345) mod 32768,
    X is B mod 12,
    N1 is N-1,
    random_list(N1,B,L).
```

The nature of the data seems highly relevant. For instance, YAP and XSB performs better with repeated data and B-Prolog performs better with random data. As can be observed from Figure 1 plots a and c, time complexity is larger than linear in all cases, but varies depending the type of data. Space consumption is linear for repeated data in XSB and YAP, but for B-Prolog it is linear regardless of the type of data. The best time complexity is observed for B-Prolog with random data but as can be observed in plot c it is still super-linear. XSB and YAP show a different pattern where the time complexity seems to be more closely coupled to space complexity. For repeated data they are more time-efficient than B-Prolog but still significantly slower than B-Prolog with random data and still distinctively super-linear.

3 A Workaround and Its Implementation in Prolog

We present here a workaround that results in $O(1)$ time and space complexity for table lookups for programs with arbitrarily large ground structured data as input arguments. A term is represented as a set of facts, each representing a subterm which is referenced by a unique integer serving as an abstract pointer. Matching related to tabling is done solely by comparison of such pointers, independently of the underlying system. The representation is given by the following predicates which all together can be understood as an abstract datatype.

`store_term(` +*ground-term*`,` *pointer*`)`
: The *ground-term* is any ground term, and the *pointer* returned is a unique reference (an integer) for that term.

`retrieve_term(` +*pointer*`,` ?*functor*`,` ?*arg-pointers-list*`)`
: Returns the functor and a list of pointers to representations of the substructures of the term represented by *pointer*.

`full_retrieve_term(` +*pointer*`,` ?*ground-term*`)`
: Returns the term represented by *pointer*.

More precisely, it must hold for any ground term s, that the query

`store_term(`s`, P), full_retrieve_term(P, S),`

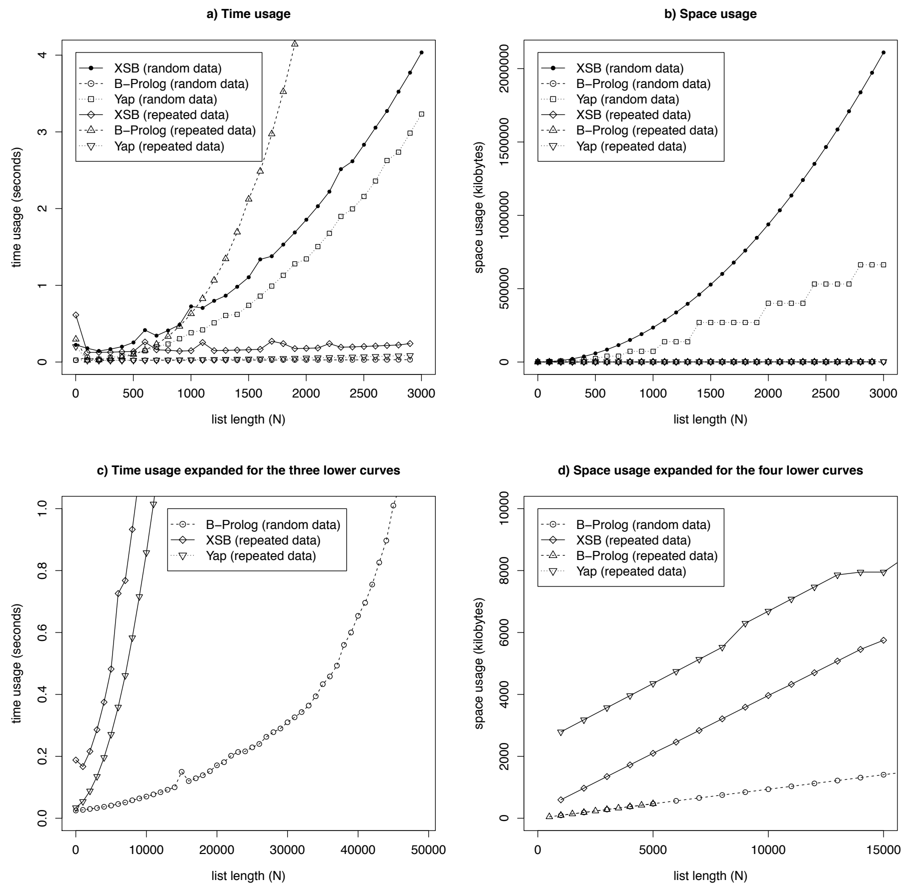

Fig. 1. Plot a) shows the time consumption of different Prolog engines for running tabled `last/2` with lists of length N and plot b) shows the table space usage. Table space usage is measured using the `statistics/1` predicate (which is different for each Prolog). In Yap it includes no specific measurement of "table space" and we measure instead the "program space" which is taken to include the table space. *Random data* means that the list contained random integers and in the *repeated data* means that the lists containing the same integer repeated N times. Plot c) and d) shows time and table space usage for the curves in a) and b) that looks flat because of the scale, but expanded for larger values of N. The curve of B-Prolog for repeated data in plot d) is truncated at 5000 due to its long running time for larger values.

assigns to the variable `S` a value identical to s. Furthermore, it must hold for any ground term s of the form $f(s_1, \ldots, s_n)$ that

store_term(s, P), retrieve_term(P, F, Ss),

assigns to the variable F the symbol f, and to `Ss` a list of ground values $[p_1, \ldots, p_n]$ such that additional queries

full_retrieve_term(p_i, S_i), $i = 1, \ldots, n$

assign to the variables S_i values identical to s_i.

Example 1. The following call converts the term `f(a,g(b))` into its internal representation and returns a pointer value in the variable `P`.

```
store_term(f(a,g(b)),P).
```

After this, the following sequence of calls will succeed.

```
retrieve_term(P,f,[P1,P2]),
retrieve_term(P1,a,[]),
retrieve_term(P2,g,[P21]),
retrieve_term(P21,b,[]),
full_retrieve_term(P,f(a,g(b))).
```

Example 2. One possible way of implementing the predicates introduced above is to have `store_term/2` asserting facts for the `retrieve_term/3` predicate using increasing integers as pointers. Then the call `store_term(f(a,g(b)),P)` considered in example 1 may assign the value `100` to `P` and as a side-effect assert the following facts.

```
retrieve_term(100,f,[101,102]).
retrieve_term(101,a,[]).
retrieve_term(102,g,[103]).
retrieve_term(103,b,[]).
```

Notice that Prolog's indexing on first arguments ensures a constant lookup time.

Example 3. In an application for which large numbers of identical subterms are expected, the representation can exploit this for sharing, so for example the term `h(very(large,sub(term)), very(large,sub(term)))` may be represented by the pointer value `200` and the following facts.

```
retrieve_term(200,g,[201,201]).
retrieve_term(201,very,[...]).
...
```

This will increase the time complexity for `store_term/2` but the advantages are *i)* storage consumption is reduced, and more importantly *ii)* an additional – and for the right sort of application programs drastic – speed-up may be obtained from the improved utilization of tabling that this automatically implies.

Finally we introduce a utility predicate which may simplify the use of the representation in application programs. It utilizes a special kind of terms, called *patterns*, which are not necessarily ground and which may contain subterms of the form `lazy(`*variable*`)`.

`lookup_pattern(` +*pointer*`,` +*pattern*`)`

The *pattern* is matched in a recursive way against the term represented by the pointer p in the following way.

- `lookup_pattern(`p`,X)` is treated as `full_retrieve_term(`p`,X)`.
- `lookup_pattern(`p`,lazy(X))` unifies `X` with p.
- For any other *pattern* `=.. [F,X`$_1$`,...,X`$_n$`]` we call
  `retrieve_term(`p`, F, [P`$_1$`,...,P`$_n$`])`
  followed by `lookup_pattern(P`$_i$`,X`$_i$`)`, $i = 1, \ldots, n$.

Example 4. Continuing example 2, we get that

```
 lookup_pattern(100, f(X,lazy(Y)))
```

leads to `X=a` and `Y=102`.

The `lookup_pattern/2` predicate simplifies the program transformation introduced in section 5 although further efficiency can be gained by compiling it out for each specific pattern.

4 Examples

The impact of indexing for ground arguments with tabled execution is evaluated through two experiments. Firstly, we compare the performance of existing Prolog systems with tabling for a simple edit distance problem. The second experiment is related to our driving motivation – biological sequence analysis in PRISM, exemplified for probabilistic inference with Hidden Markov Models. All experiments were run on a MacBook Pro with a 2.53 GHz Intel core 2 Duo processor, 4 GB memory and Mac OS X version 10.6.8 (Snow Leopard).

4.1 Example: Edit Distance

We consider a minimal edit-distance algorithm written in Prolog which is dependent on tabling for any non-trivial problem. Time and space consumption are measured for increasing problem sizes in the three major tabling systems with and without our indexed representation.

Edit-distance is the textbook example dynamic programming. In the classic imperative formulation of the problem, a matrix with N^2 values is calculated, such that the calculation of the value for each cell is a constant time operation that depends on at most three other cells. The theoretical best time complexity of edit distance has been proven to be $O(N^2)$ [16]. Dynamic programming problems exhibit *optimal sub-structure* which implies that partial solutions can be reused rather than recomputed [1]. Tabling supports dynamic programming since resolved goals are kept in a table and reused rather than re-derived if the tabled goals are called again. The following Prolog program implements *minimal edit distance* between two lists; given two lists L_1 and L_2, the call `edit(`L_1`, `L_2`,D)` will return the minimal number of edits (substitutions,insertions and deletions) needed to change one of the lists into the other.

```
:- table edit/3.

edit([],[],0).

edit([],[Y|Ys],Dist) :-
   edit([],Ys,Dist1),
   Dist is 1 + Dist1.

edit([X|Xs],[],Dist) :-
   edit(Xs,[],Dist1),
   Dist is 1 + Dist1.

edit([X|Xs],[Y|Ys],Dist) :-
   edit([X|Xs],Ys,InsDist),
   edit(Xs,[Y|Ys],DelDist),
   edit(Xs,Ys,TailDist),
   (X==Y ->
       Dist = TailDist
       ;
       % Minimum of insertion, deletion or substitution
       sort([InsDist,DelDist,TailDist],[MinDist|_]),
       Dist is 1 + MinDist).
```

Without tabling the `edit/3` predicate, the same subgoals are derived again and again leading to exponential blowup, but it can be shown that the number of distinct calls are quadratic, which is the actual complexity we may hope for with optimal tabling.

The program has been transformed manually for this experiment based on the pointer based representation shown in example 2 above, simplified slightly for lists. The `retrieve_term` predicate is applied to resolve pointers during program execution. For completeness, we include a suitable implementation of `store_term/2` and `retrieve_term/2`.

```
store_term([],Index) :- assert(retrieve_term([],Index)).

store_term([X|Xs],Idx) :-
   Idx1 is Idx + 1,
   assert(retrieve_term(Idx,[X,Idx1])),
   store_term(Xs,Idx1).
```

The transformed version of the edit distance program is now as follows.

```
:- table edit/3.

edit(XIdx,YIdx,0) :-
   retrieve_term(XIdx,[]),
   retrieve_term(YIdx,[]).

edit(XIdx,YIdx,Dist) :-
   retrieve_term(XIdx,[]),
   retrieve_term(YIdx,[_,YIdxNext]),
   edit(XIdx,YIdxNext,Dist1),
   Dist is Dist1 + 1.

edit(XIdx,YIdx,Dist) :-
   retrieve_term(YIdx,[]),
   retrieve_term(XIdx,[_,XIdxNext]),
   edit(XIdxNext,YIdx,Dist1),
   Dist is Dist1 + 1.

edit(XIdx,YIdx,Dist) :-
   retrieve_term(XIdx,[X,NextXIdx]),
   retrieve_term(YIdx,[Y,NextYIdx]),
   edit(XIdx,NextYIdx,InsDist),
   edit(NextXIdx,YIdx,DelDist),
   edit(NextXIdx,NextYIdx,TailDist),
   (X==Y ->
       Dist = TailDist
       ;
       sort([InsDist,DelDist,TailDist],[MinDist|_]),
       Dist is 1 + MinDist).
```

The program is tested for randomly generated sequences of increasing lengths. We measure the total time for the different Prolog engines to load the program file, generate two different random sequences of a particular length, assert these lists using `store_term/2` and compute edit distance between these sequences, as follows.

```
run(N) :-
   random_list(N,117,L1), % Generate random list L1 with seed 117
   random_list(N,42,L2),  % Generate random list L1 with seed 42
   store_term(L1,P1),
   store_term(L2,P2),
   edit(P1,P2,_Dist).
```

The results, shown in Figure 2, demonstrate that all tested Prolog systems use more time for the unmodified tabled edit distance program than for the transformed program when applied to large data instances. For XSB and Yap the major factor impacting time complexity seems to be space consumption. The transformation has a positive impact space complexity regardless of the underlying tabling strategy. For B-Prolog, space consumption is much closer to the theoretical $O(N^2)$. Even though B-Prolog is very space efficient, the transformed program still uses less memory.

For larger problem instances the transformation has a significant impact on time complexity. XSB seems to benefit greatly from the transformation, although it starts out the slowest, it catches up for longer sequences, where it outperforms the two other Prologs in time efficiency. Yap seems to gain a modest boost from the transformation strategy and still seems to have a rather high time complexity although it is significantly faster than without the transformation. For B-Prolog, the two versions perform more or less the same for sequences of length up to 350, but for longer sequences (not shown in the figure) the transformed version is significantly faster: for example, with length 1000, the execution times are 7.5 seconds for the transformed and 21.5 seconds for the original version.

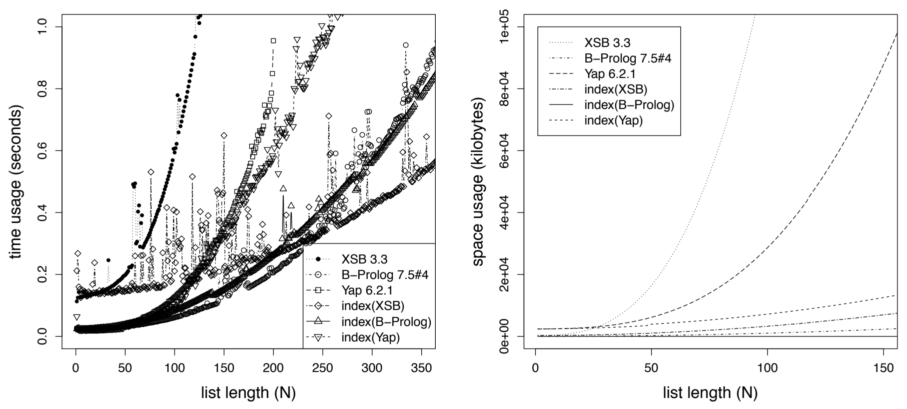

Fig. 2. The first plot shows the time consumption of different Prolog engines for edit distance with two lists of length N. The second is a plot of the space consumption for the same calls. The plots for both normal tabled in execution and execution of a transformed program that uses indexing as described in section 3 are shown, e.g. the plot $index(X)$ shows the performance of the transformed version for Prolog implementation X.

4.2 Example: Hidden Markov Model in PRISM

PRISM [8] is an extension of Prolog with special goals representing random variables. A global declaration such as `values(coin,[head,tail])` introduces a so-called multivalued switch which means that an occurrence of the subgoal `msw(coin,C)` represents a probabilistic choice of assigning either `head` or `tail` to C. The semantics of PRISM is defined in terms of probabilistic Herbrand models, which means that a program specifies a probability of any goal G to be true determined from the possible combinations of msw outcomes that happen to make G true.

The PRISM system supports various probabilistic inferences, such as finding an optimal derivation, computing the probability for a goal or deriving msw probabilities by learning from a set of goals. The algorithms behind these inferences are dynamic programming algorithms and PRISM is implemented in B-Prolog [17], relying heavily on tabling for the efficiency of the probabilistic inferences.

We consider the example of a Hidden Markov Model (HMM) in PRISM taken from the PRISM manual [10] and adapted here to accommodate variable length sequences. In general, an HMM is a probabilistic model for sequential phenomena based on a finite automaton, which chooses state transitions and emissions by probabilistic choices; see [5] for a general introduction to HMMs and [3] for an account on how different HMMs are expressed in PRISM. Our example program is the following.

```
values(init,[s0,s1]).                hmm(_,[]).
values(out(_),[a,b]).
values(tr(_),[s0,s1]).               hmm(S,[Ob|Y]) :-
                                         msw(out(S),Ob),
hmm(L):-                                 msw(tr(S),Next),
    msw(init,S),                         hmm(Next,Y).
    hmm(S,L).
```

The init, out($-$) and tr($-$) switches determine initial state, state transitions and emissions. Notice that two last ones are parameterized meaning that they define a switch for whatever value is substituted in for the parameter, which in this program always is the present state.

Using the same list encoding as in the previous example, the recursive predicate is rewritten as follows.

```
hmm(S,ObsPtr):-
   retrieve_term(ObsPtr,[]).

hmm(S,ObsPtr) :-
   retrieve_term(ObsPtr,[Ob,Y]),
   msw(out(S),Ob),
   msw(tr(S),Next),
   hmm(Next,Y).
```

The rewritten program can be shown to be semantics preserving wrt a standard Prolog semantics as well as PRISM's probabilistic semantics, and thus running PRISM's utilities for probability calculations should yield the same results.

When calculating the probability of a given goal, PRISM iterates over all possible ways to execute the goal using tabling to avoid enumerating the exponential number of different derivations. The same principle applies for PRISMs version of the Viterbi algorithm which is a dynamic programming algorithm that finds the most probable derivation. Assuming optimal execution of tabling, these algorithms should in principle run in linear time.

We measured running times of probability calculations (`prob` in PRISM lingo) for both the original and the transformed version of the PRISM HMM program with sequences of increasing lengths from 100 to 5000. The actual sequences used are instances of the pattern `[a,b]` repeated a number of times. The results are shown in Figure 3. It is apparent from the figure that indexed lookups results in approximately linear running time while the running time is at least quadratic for the unmodified program. The reported times are measured using `prism_statistics(infer_time,Time)`, which is a PRISM built-in predicate.

We did not measure running times of sequences longer than 5000 for the unmodified program, but the transformed program scales up to sequences much longer than this, for instance, the time for probability calculation for a sequence of length 100000 takes less than 5 seconds.

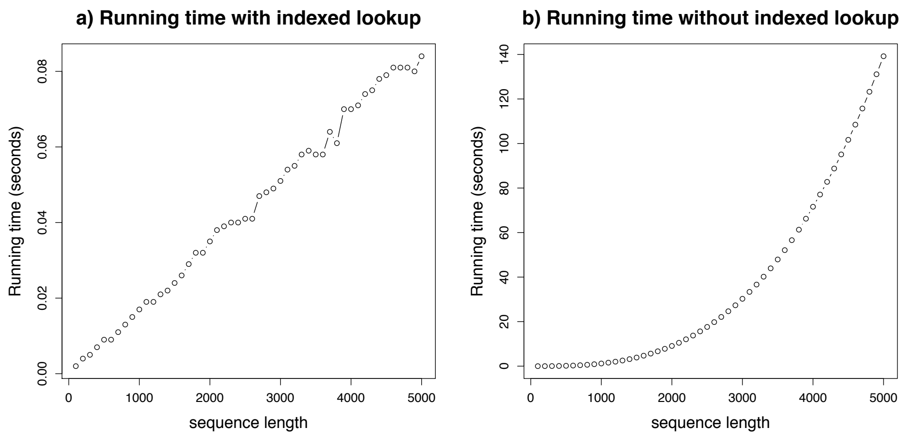

Fig. 3. The running time of the a) the transformed PRISM program and b) the unmodified PRISM program. While a) shows a linear development as function of sequence length, the development in b) is a higher-degree polynomial. Notice also the different scales on the vertical axes.

5 Automatic Program Transformation

The indexed versions of the example programs shown in section 4 can be produced automatically by a straightforward program transformation. The user must declare modes for which arguments predicate arguments that should be indexed. For the HMM program of section 4.2 this may look as follows; plus means transform the argument, minus means keep it unchanged.

```
table_index_mode(hmm(+))
table_index_mode(hmm(-,+))
```

The correctness of the transformation depends on the following properties of the program.

- The arguments indicated for indexing must be called with ground data only.
- Variables that occur in an indexed argument in the head of a clause, cannot occur in the body of that clause in both an indexed and a non-indexed argument.
- Any argument of a goal within a clause body which is declared to be indexed, must be given as a variable that also occurs in an indexed argument in the head of that clause.

Each clause whose head predicate is covered by a table mode declaration is transformed using the procedure outlined in algorithm 1, and all other clauses are left untouched. The transformation moves any term appearing in an indexed

```
for each clause H:-B in original program do
    if table_index_mode (M) matching H then
        for each argument $H_i \in H, M_i \in M$ do
            if $M_i =$ '+' then
                $H'_i \leftarrow \mathtt{MarkLazy}(H_i, B)$
                $B \leftarrow (lookup\_pattern(V_i, H'_i), B)$
                $H_i \leftarrow V_i$
        end
end
```

where MarkLazy is defined as

```
MarkLazy($H_i$,B) :
  $PotentialLazy =$ variables in all goals $G \in$ B
    where $G$ has table_index_mode declaration
  $NonLazy =$ variables in all goals $G \in$ B
    where $G$ has no table_index_mode declaration
  $Lazy = PotentialLazy \setminus NonLazy$
  for each variable $V \in H_i$ do
      if $V \in Lazy$ then
          $V \leftarrow lazy(V)$
  end
```

Algorithm 1: Program transformation.

position in the head of a clause into a call to the lookup_pattern predicate, which is added to the body. Variables in such terms are marked lazy when they do not occur in any non-indexed argument inside the clause. This transformation can be shown to be semantics preserving for programs satisfying the requirements given above.

The translation can be further enhanced by an unfolding of `lookup_pattern` calls into specialized calls to `retrieve_term` as shown in the examples in the previous section. This last step gave a speed-up of a factor of 5 for these examples when comparing with implementations using `lookup_pattern` directly.

6 Limitations

Our transformation assumes ground input arguments. As illustrated by the examples, this has applications to a lot of interesting problems, in particular dynamic programming problems. With regard to PRISM, our transformation is useful for ordinary probability calculation, Viterbi decoding and supervised learning. For other probabilistic inferences such as sampling, posterior decoding, unsupervised and semi-supervised learning, arguments containing variables are required. Sampling is of minor concern, since this can be done in linear time using the original program.

We currently have no optimization for structured terms in output arguments – they must be handled by the usual tabling mechanism. Structured terms in output arguments have the same consequences for complexity, which can be observed for instance with the well-known `append/3` predicate. Suppose that `append/3` is tabled and transformed using our approach, e.g. with `table_index_mode(+,+,-)`. Using our workaround, the space complexity for the input lists will be kept linear rather than quadratic, but the answers for the third list is tabled in the usual way which leads to quadratic space complexity nevertheless. Output arguments that do not contain structured data — as in the case of edit distance — do not present such a problem since the output argument is of constant size.

A drawback of our transformation is that it, by replacing the patterns in the head of rules with pointers, circumvents Prolog own indexing mechanism. As result, indexing cannot use the pattern of the arguments to determine which clauses to try. Instead, when multiple clauses with same name and arity exist, Prolog will have to try each of them in order and creates a choice point each time it tries a clause. This adds a constant factor — corresponding to the number of such clauses — to the running time of the program. It most practical programs it is realistic to assume that this factor will be fairly low, e.g. in the edit distance program only four such clauses exist.

7 Related work

The hashing employed by B-Prolog and the global trie of YAP [6] address a related problem. Both methods reduce space consumption and this may lead to reductions in running time since less copying is needed. However, even with these mechanisms complexity is sub-optimal as shown in section 2. Furthermore, the methods have the drawback that the running time depends on the type of data. In comparison, our approach is data invariant and yields optimal complexity.

Due to restrictions in the Mercury language, input arguments are always ground, and the tabling system provides an option which identifies arguments by their pointers [11] (see also more detailed explanations in the reference manual [4]). This yields constant time storing and comparison of tabled arguments, similar to how any standard tabling mechanism will work for the programs produced by our program transformation.

The problem with tabling of structured data has addressed in applications with methods similar to our approach. In particular, in chart-parsing with DCGs supported by tabling, position indexed facts has been used [12]. A similar approach has been applied to PCFG parsing in PRISM [9]. This works by splitting the input list, $t_1 \ldots t_N$ into facts, $\{$`pos(`$1, t_1, 2$`)`, ..., `pos(`$N-1, t_n, N$`)`$\}$. XSB Prolog have special constructions for tabled DCGs, where the standard `'C'/3'` predicate is replaced by a special version that instead of using difference lists, utilize position indexed facts constructed from the original input list [14]. The position indexed difference list approach is quite similar to our approach, but is specific for difference lists. Our approach is more generally applicable and can be used with various kinds of structured data.

8 Conclusion

We have demonstrated that major Prolog implementations do not efficiently handle tabling of structured data and we have provided a program transformation that ensures $O(1)$ time and space complexity of tabled lookups of goals with structured data in input arguments and is applicable regardless of inefficiencies with structured data in the underlying tabling implementation. We have demonstrated the applicability of our transformation using examples from dynamic programming in Prolog and PRISM. The transformation makes it possible to scale to much larger problem instances.

Our program transformation should be seen as workaround, until such optimizations find their way into the tabling systems. We hope that Prolog implementors will pick up on this and integrate such optimizations directly in the tabling systems, so that the user does not need to transform his program, and need not worry about the underlying tabled representation and its implicit complexity.

Acknowledgements. This work is supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

We would like to thank Neng-Fa Zhou and Yoshitaka Kameya for their encouraging comments on a very early draft of this paper. We would also like to thank to the anonymous reviewers for insightful and constructive reviews.

References

1. R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.
2. H. Christiansen and J. P. Gallagher. Non-discriminating arguments and their uses. In P. M. Hill and D. S. Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2009.
3. H. Christiansen, C. T. Have, O. T. Lassen, and M. Petit. Taming the zoo of discrete HMM subspecies & some of their relatives. In *Biology, Computation and Linguistics, New Interdisciplinary Paradigms*, volume 228 of *Frontiers in Artificial Intelligence and Applications*, pages 28–42. IOS Press, 2011.
4. F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, C. Speirs, T. Dowd, R. Becket, M. Brown, and P. Wang. *The Mercury Language Reference Manual. Version 11.01*, 2011. Available at http://www.mercury.cs.mu.oz.au/information/documentation.html.
5. L.R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
6. J. Raimundo and R. Rocha. Global Trie for Subterms. In S. Abreu and V. S. Costa, editors, *Proceedings of the 11th Colloquium on Implementation of Constraint and LOgic Programming Systems, CICLOPS'2011*, pages 34–48, Lexington, Kentucky, USA, July 2011.
7. R. Rocha, F. Silva, and V. S. Costa. A tabling engine for the Yap Prolog system. In *Proceedings of the 2000 APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'00)*, La Habana, Cuba, December 2000.
8. T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *IJCAI*, pages 1330–1339, 1997.
9. T. Sato and Y. Kameya. New advances in logic-based probabilistic modeling, 2007.
10. T. Sato, N.-F. Zhou, Y. Kameya, and Y. Izumi. *PRISM User's Manual (Version 2.0)*, 2010.
11. Z. Somogyi and K. F. Sagonas. Tabling in mercury: Design and implementation. In P. V. Hentenryck, editor, *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 of *Lecture Notes in Computer Science*, pages 150–167. Springer, 2006.
12. T. Swift. Design patterns for tabled logic programming. In S. Abreu and D. Seipel, editors, *INAP*, volume 6547 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2009.
13. T. Swift and D. S. Warren. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 2011. To appear.
14. T. Swift and D. S. Warren. *The XSB Programmer's Manual. Version 3.3*, June 2011.
15. H. Tamaki and T. Sato. OLD resolution with tabulation. In E. Y. Shapiro, editor, *ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.
16. C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *J. ACM*, 23(1):13–16, 1976.
17. N.-F. Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 2011. To appear.
18. N.-F. Zhou, Y.-D. Shen, L.-Y. Yuan, and J.-H. You. Implementation of a linear tabling mechanism. In E. Pontelli and V. S. Costa, editors, *PADL*, volume 1753 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2000.