Towards Pre-Indexed Terms*

J. F. Morales[1] and M. Hermenegildo[1,2]

[1] IMDEA Software Research Insitute, Madrid, Spain
[2] School of Computer Science, Technical University of Madrid, Spain

Abstract. Indexing of terms and clauses is a well-known technique used in Prolog implementations (as well as automated theorem provers) to speed up search. In this paper we show how the same mechanism can be used to implement efficient reversible mappings between different term representations, which we call *pre-indexing*s. Based on user-provided term descriptions, these mappings allow us to use more efficient data encodings internally, such as prefix trees. We show that for some classes of programs, we can drastically improve the efficiency by applying such mappings at selected program points.

1 Introduction

Terms are the most important data type for languages and systems based on first-order logic, such as (constraint) logic programming or resolution-based automated theorem provers. Terms are inductively defined as variables, atoms, numbers, and compound terms (or structures) comprised by a functor and a sequence of terms.[3] Two main representations for Prolog terms have been proposed. Early Prolog systems, such as the Marseille and DEC-10 implementations, used *structure sharing* [2], while the WAM [13,1] –and consequently most modern Prolog implementations– uses *structure copying*. In structure sharing, terms are represented as a pair of pointers, one for the structure skeleton, which is shared among several instances, and another for the binding environment, which determines a particular instantiation. In contrast, structure copying makes a copy of the structure for each newly created term. The encoding of terms in memory resembles tree-like data structures.

In order to speed up resolution, sophisticated term indexing has been implemented both in Prolog [1,8] and automated theorem provers [7]. By using specialized data structures (such as, e.g., tries), indexing achieves sub-linear complexity in clause selection. Similar techniques are used to efficiently store predicate solutions in tabling [11]. This efficient machinery for indexing is often

* Research supported in part by projects EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2008-05624 *DOVES*, and Comunidad de Madrid TIC/1465 *PROMETIDOS-CM*.

[3] Additionally, many Prolog systems implement an extension mechanism for variable domains using *attributed variables*.

attractive for storing and manipulating program data, such as dynamic predicates. Indexed dynamic predicates offer the benefits of efficient key-value data structures while hiding the implementation details from the user program.

Modulo some issues like variable sharing, there is thus a duality in programming style between *explicitly* encoding data as terms or encoding data *implicitly* as tuples in dynamic predicates. However, although both alternatives have some *declarative* flavor, it is also frequent to find code where, for performance reasons, the data is represented in the end in a quite unnatural way. E.g., the set $\{1, 2, 3, \ldots, n\}$ can be represented naturally as the term `[1,2,3,...,n]` (equivalent to a linked list). However, depending on the lifetime and operations to be performed on the data, binary trees, some other map-like structure, or dynamic predicates may be preferable. These changes in representation often propagate through the whole program.

The goal of this paper is to study the merits of term indexing during term creation rather than at clause selection time. We exploit the fact that data has frequently a fixed skeleton structure, and introduce a mapping in order to index and share that part. This mapping is derived from program declarations specifying term encoding (called *rtype*s, for r*epresentation types*) and annotations defining the program points where *pre-indexing* of terms is performed. This is done on top of structure copying, so that no large changes are required in a typical Prolog runtime system. Moreover, the approach does not require large changes in program structure, which makes *rtype*s easily interchangeable.

We have implemented a prototype as a Ciao package that deals with *rtype* declarations as well as with some additional syntactic sugar that we provide for marking pre-indexing points. We leave as future work the automatic selection of encoding decisions based on profiling and more detailed cost models.

2 Background

We follow the definitions and naming conventions for *term indexing* of [4,7]. Given a set of terms $\mathcal{L}$ (the *indexed terms*), a binary relation R over terms (the *retrieval condition*), and a term t (the *query term*), we want to identify the subset $\mathcal{M} \subseteq \mathcal{L}$ consisting of all the terms l such that $R(l, t)$ holds (i.e., such that l is R-compatible with t). We are interested in the following retrieval conditions R (where σ is a substitution):

- $\mathit{unif}(l, t) \Leftrightarrow \exists\sigma\ l\sigma = t\sigma$ (unification)
- $\mathit{inst}(l, t) \Leftrightarrow \exists\sigma\ l = t\sigma$ (instance check)
- $\mathit{gen}(l, t) \Leftrightarrow \exists\sigma\ l\sigma = t$ (generalization check)
- $\mathit{variant}(l, t) \Leftrightarrow \exists\sigma\ l\sigma = t$ and σ is a renaming substitution (variant check)

Example 1. Given $L = \{h(f(A)), h(f(B, C)), h(g(D))\}$, $t = h(f(1))$, and $R = \mathit{unif}$, then $M = \{h(f(A))\}$.

The objective of *term indexing* is to implement fast retrieval of candidate terms. This is done by processing the indexed set $\mathcal{L}$ into specialized data structures (*index construction*) and modifying this index when terms are inserted or deleted from $\mathcal{L}$ (*index maintenance*).

When the retrieval condition makes use of the function symbols in the query and indexed terms, it is called *function symbol based indexing*.

In Prolog, indexing finds the set of program clauses such that their heads unify with a given literal in the goal. In tabled logic programming, this is also interesting for detecting if a new goal is a variant or subsumed by a previously evaluated subgoal [6,10].

Limitations of indexing. Depending on the part of the terms that is indexed and the supporting data structure, the worst case cost of indexing is proportional to the size of the term. When computing hash keys, the whole term needs to be traversed (e.g., computing the key for `h(f(A))` requires walking over `h` and `f`). This may be prohibitively costly, not only in the maintenance of the indices, but also in the lookup. As a compromise many systems rely only on first argument, first level indexing (with constant hash table lookup, relying on linear search for the selected clauses). However, when the application needs stronger, multi-level indexing, lookup costs are repeated many times for each clause selection operation.

3 Pre-indexing

The goal of pre-indexing is to move lookup costs to term building time. The idea that we propose herein is to use a bijective mapping between the standard and the pre-indexed representations of terms, at selected program points. The fact that terms can be partially instantiated brings in a practical problem, since bounding a variable may affect many precomputed indices (e.g., precomputed indices for `H=h(X)`, `G=g(X)` may need a change after `X=1`). Our proposed solution to this problem is to restrict the mapping to terms of a specific form, based on (herein, user-provided) *instantiation types*.

Definition 1 (Instantiation type). *We say that t is an instance of an* instantiation type τ *(defined as a unary predicate), written as check*$(\tau(t))$*, if there exists a term l such that $\tau(l)$ is in the model of τ and gen*(l,t) *(or inst*(t,l)*).*

For conciseness, we will describe the restricted form of instantiation types used herein using a specialized syntax: [4]

```
:- rtype lst ---> [] ; [any|lst]
```

In these rules `any` represents any term or variable while `nv` represents any `nonvar` term. The rule above thus corresponds to the predicate:

```
lst([]).
lst([_|Xs]) :- lst(Xs).
```

[4] Despite the syntax being similar to that described in [9], note that the semantics is not equivalent.

Example 2. According to the definition above for `lst`, the terms `[1,2,3]` and `[_,2]` belong to `lst` while `[1|_]` does not. If `nv` were used instead of `any` in the definition above then `[_,2]` would also not belong to `lst`.

Type-based pre-indexing. The idea behind pre-indexing is to maintain specialized indexing structures for each *rtype* (which in this work is done based on user annotations). Conceptually, the indexing structure will keep track of all the *rtype* inhabitants dynamically, assigning a unique identifier (the pre-index key) to each of them. E.g., for `lst` we could assign $\{$ `[]` $\mapsto k_0$, `[_]` $\mapsto k_1$, `[_,_]` $\mapsto k_2, \ldots\}$.

Translation between pre-indexed and non-pre-indexed forms is defined in terms of a *pre-indexing casting*. Given $check(\tau(t))$, $\exists\ l \in |\tau|$ (set of "weakest" terms for which τ holds) such that $gen(l, t)$.

Definition 2 (Pre-indexing cast). *A* pre-indexing cast *of type τ is a bijective mapping between terms, denoted by $\#\tau$, with the following properties:*

- *For every term x and substitution σ so that $check(\tau(x))$, then $\#\tau(x\sigma) = \#\tau(x)\sigma$ (σ-commutative), and*
- *the first-level functor of $\#\tau(x)$* encodes *the structure of the arguments (so that it uniquely identifies the* rtype *inhabitant).*

Informally, the first property ensures that pre-indexing casts can be selectively introduced in a program without altering the (substitution) semantics. Moreover, the meaning of many built-ins is also preserved after pre-indexing, as expressed in the following theorem.

Theorem 1 (Built-in homomorphism). *Given $check(\tau(x))$ and $check(\tau(y))$, then $unif(x, y) \Leftrightarrow unif(\#\tau(x), \#\tau(y))$ (equivalently for gen, inst, variant, and other built-ins like `==/2`, `ground/1`).*

Proof. $unif(x, y) \Leftrightarrow$ [def. of unif] $\exists\sigma\ x\sigma = y\sigma$. Since $\#\tau$ is bijective, then $\#\tau(x\sigma) = \#\tau(y\sigma) \Leftrightarrow$ [σ-commutative] $\#\tau(x)\sigma = \#\tau(y)\sigma$. Given the def. of *unif*, it follows that $\mathrm{unif}(\#\tau(x), \#\tau(y))$. The proofs for other built-ins are similar.

In this work we do not require the semantics of built-ins like `@<` (i.e., *term ordering*) to be preserved, but if desired this can be achieved by selecting carefully the order of keys in the pre-indexed term. Similarly, functor arity in principle will not be preserved since ground arguments that are part of the *rtype* structure are allowed to be removed.

3.1 Building pre-indexed terms

We are interested in building terms directly into their pre-indexed form. To achieve this we take inspiration from WAM compilation. Complex terms in variable-term unifications are decomposed into simple variable-structure unifications $X = f(A_1, \ldots, A_n)$ where all the A_i are variables. In WAM bytecode, this is further decomposed into a `put_str` f/n (or `get_str` f/n) instruction followed by a sequence of `unify_arg` A_i. These instructions can be expressed as follows:

```
put_str(X,F/N,S0,S1),     % | F/N |
unify_arg(A1,S1,S2)       % | F/N | A1 |
...
unify_arg(An,Sn,S)        % | F/N | A1 | ... | An |
```

where the S_i represent each intermediate heap state, which is illustrated in the comments on the right.

Assume that each argument A_i can be split into its indexed part A_ik and its value part A_iv (which may omit information present in the key). *Pre-indexing* builds terms that encode A_ik into the main functor:

```
g_put_str(X,F/N,S0,S1),     % | F/N |
g_unify_arg(A1,S1,S2)       % | F/N<A1k> | A1v |
...
g_unify_arg(An,Sn,S)        % | F/N<A1k,...,Ank> | A1v | ... | Anv |
```

The *rtype* constructor annotations (that we will see in Section 3.2) indicate how the functor and arguments are indexed.

Cost analysis. Building and unifying pre-indexed terms have impact both on performance and memory usage. First, regarding time, although pre-indexing operations can be slower, clause selection becomes faster, as it avoids repetitive lookups on the fixed structure of terms. In the best case, $O(n)$ lookups (where n is the size of the term) become $O(1)$. Other operations like unification are sped-up (e.g., earlier failure if keys are different). Second, pre-indexing has an impact on memory usage. Exploiting the data structure allows more compact representations, e.g., `bitpair(bool,bool)` can be assigned an integer as key (without storage costs). In other cases, the supporting *index structures* may effectively share the common part of terms (at the cost of maintaining those structures).

3.2 Pre-indexing Methods

Pre-indexing is enabled in an *rtype* by annotating each constructor with modifiers that specify the *indexing method*. Currently we support compact trie-like representations and packaged integer encodings.

Trie representation is specified with the `index(Args)` modifier, which indicates the order in which arguments are walked in the decision-tree. The process is similar to term creation in the heap, but instead of moving a heap pointer, we combine it with walking through a trie of nodes. Keys are retrieved from the term part that corresponds to the *rtype* structure.

For example, let us consider the input set of terms $[a(x), c(z)]$, $[a(x), d(w)]$, $[b(y), c(z)]$, $[b(y), d(w)]$, where a, b, c, d are function symbols and x, y, z, w are

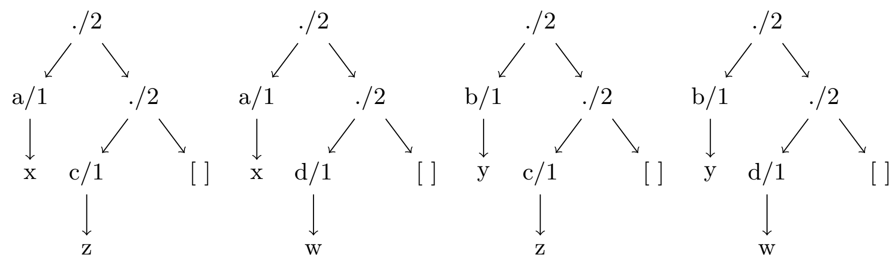

Fig. 1. Example terms for pre-indexing

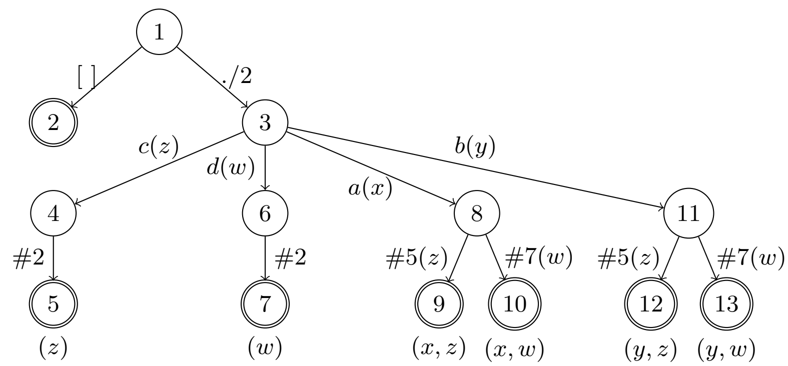

Fig. 2. Index for example terms (*rtype* `lst ---> [] ; [nv|lst]:::index([0,1,2])`)

variable symbols. The heap representation is shown in Fig. 1.[5] We will compare different *rtype* definitions for representing these terms.

As mentioned before, `nv` represents the *rtype* for any `nonvar` term (where its first level functor is part of the type). The declaration:

```
:- rtype lst ---> [] ; [nv|lst]:::index([0,1,2]).
```

specifies that the lookup order for `[_|_]` is a) the constructor name (*./2*), b) the first argument (not pre-indexed), and c) the second argument (pre-indexed). The resulting trie is in Fig. 2. In the figure, each node number represents a position in the trie. Singly circled nodes are temporary nodes, doubly circled nodes are final nodes. Final nodes encode terms. The initial node (*#1*) is unique for each *rtype*. Labels between nodes indicate the lookup input. They can be constructor names (e.g., *./2*), `nv` terms (e.g., *b(y)*), or other pre-indexed `lst` (e.g., *#2* for *[]*, or *#5(z)* for *[c(z)]*). The arguments are placeholders for the non-indexed information. That is, a term *[a(g),c(h)]* would be encoded as *#9(g,h)*.

Trie indexing also supports *anchoring* on non-root nodes. Consider this declaration:

[5] Remember that `[1,2] = .(1,.(2,[]))`.

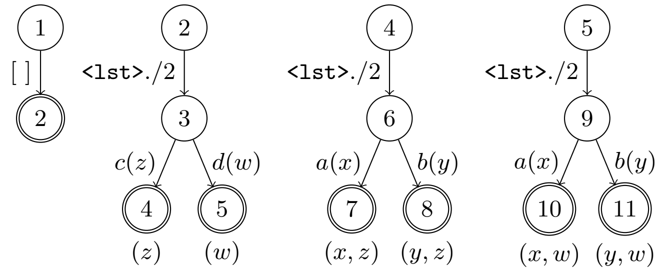

Fig. 3. Index for example terms (*rtype* `lst ---> [] ; [nv|lst]:::index([2,0,1])`)

```
:- rtype lst ---> [] ; [nv|lst]:::index([2,0,1]).
```

Figure 3 shows the resulting trie. The lookup now starts from the second argument, then the constructor name, and finally the first argument. The main difference w.r.t. the previous indexing method is that the beginning node is another pre-indexed term. This may lead to more optimal memory layouts and need fewer lookup operations. Note that constructor names in the edges from initial nodes need to be prefixed with the name of the *rtype*. This is necessary to avoid ambiguities, since the initial node is no longer unique.

Garbage Collection and Indexing Methods. Indexing structures require special treatment for garbage collection.[6] In principle, it would not be necessary to keep in a trie nodes for terms that are no longer reachable (e.g., from the heap, WAM registers, or dynamic predicates), except for caching to speed-up node creation. Node removal may make use of lookup order. That is, if a key at a temporary level n corresponds to an atom that is no longer reachable, then all nodes above n can be safely discarded.

Anchoring on non-root nodes allows the simulation of interesting memory layouts. For example, a simple way to encode objects in Prolog is by introducing a *new object* operation that creates new fresh atoms, and storing object attributes with a dynamic `objattr(ObjId, AttrName, AttrValue)` predicate. Anchoring on `ObjId` allows fast deletion (at the implementation level) of all attributes of a specific object when it becomes unreachable.

4 Applications and Experimental Evaluation

To show the feasibility of the approach, we have implemented the pre-indexing transformations as source-to-source transformations within the Ciao system. This is done within a Ciao package which defines the syntax and processes the *rtype* declarations as well as the marking of pre-indexing points.

[6] Automatic garbage collection of indexing structures is not supported in the current implementation.

```
compress(Cs, Result) :-               % Compress Cs
    build_dict(256),                  % Build the dictionary
    compress_(Cs, #lst([]), Result).

compress_([], W, [I]) :-              % Empty, output code for W
    dict(W,I).
compress_([C|Cs], W, Result) :-       % Compress C
    WC = #lst([C|^W]),
    ( dict(WC,_) ->                   % WC is in dictionary
        W2 = WC,
        Result = Result0
    ; dict(W,I),                      % WC not in dictionary
      Result = [I|Result0],           % Output the code for W
      insert(WC),                     % Add WC to the dictionary
      W2 = #lst([C])
    ),
    compress_(Cs, W2, Result0).
```

Fig. 4. LZW Compression: Main code.

As examples, we show algorithmically efficient implementations of the Lempel-Ziv-Welch (LZW) lossless data compression algorithm and the Floyd-Warshall algorithm for finding the shortest paths in a weighted graph, as well as some considerations regarding supporting module system implementation. In the following code, `forall/2` is defined as `\+ (Cond, \+ Goal)`.

4.1 Lempel-Ziv-Welch compression

Lempel-Ziv-Welch (LZW) [14] is a lossless data compression algorithm. It encodes an input string by building an indexed dictionary D of words and writing a list of dictionary indices, as follows:

1- $D := \{w \mid w \text{ has length } 1\}$ (all strings of length one).
2- Remove from input the longest prefix that matches some word W in D, and emit its dictionary index.
3- Read new character C, $D := D \cup \mathsf{concat}(W, C)$, go to step 2; otherwise, stop.

A simple Prolog implementation is shown in Fig. 4 and Fig. 5. Our implementation uses a *dynamic* predicate `dict/2` to store words and corresponding numeric indices (for output). Step 1 is implemented in the `build_dict/1` predicate. Steps 2 and 3 are implemented in the `compress_/3` predicate. For encoding words we use lists. We are only interested in adding new characters and word matching. For that, list construction and unification are good enough. We keep words in reverse order so that appending a character is done in constant time. For constant-time matching, we use an *rtype* for pre-indexing lists. The implementation is straighforward. Note that we add a character to a word in `WC =`

```
1   % Mapping between words and dictionary index
2   :- data dict/2.
3
4   % NOTE: #lst can be changed or removed, ^ escapes cast
5   % Anchors to 2nd arg in constructor
6   :- rtype lst ---> [] ; [int|lst]:::index([2,0,1]).
7
8   build_dict(Size) :-                % Initial dictionary
9       assertz(dictsize(Size)),
10      Size1 is Size - 1,
11      forall(between(0, Size1, I),   % Single code entry for I
12        assertz(dict(#lst([I]), I))).
13
14  insert(W) :-                       % Add W to the dictionary
15      retract(dictsize(Size)), Size1 is Size + 1, assertz(dictsize(Size1)),
16      assertz(dict(W, Size)).
```

Fig. 5. LZW Compression: Auxiliary code and *rtype* definition for words.

	data size		indexing (time)		
	original	result	none	clause	term
data1	1326	732	0.074	0.025	0.015
data2	83101	20340	49.350	1.231	0.458
data3	149117	18859	93.178	2.566	0.524

Table 1. Performance of LZW compression (in seconds) by indexing method.

`#lst([C|^W])` (Line 8). The annotation indicates that words are pre-indexed using the `lst` *rtype* and that `W` is already pre-indexed (indicated by the escape `^` prefix). Thus we can effectively obtain optimal algorithmic complexity.

Performance evaluation. We have encoded three files of different format and size (two HTML files and a Ciao bytecode object) and measured the performance of alternative indexing and pre-indexing options. The experimental results for the algorithm implementation are shown in Table 1.[7] The columns under *indexing* show the execution time in seconds for different indexing methods: *none* indicates that no indexing is used (except for the default first argument, first level indexing); *clause* performs multi-level indexing on `dict/2`; *term* uses pre-indexed terms.

Clearly, disabling indexing performs badly as the number of entries in the dictionary grows, since it requires one linear (w.r.t. the dictionary size) lookup operation for each input code. Clause indexing reduces lookup complexity and shows a much improved performance. Still, the cost has a linear factor w.r.t. the

[7] Despite the simplicity of the implementation, we obtain compression rates similar to `gzip`.

word size. Term pre-indexing is the faster implementation, since the linear factor has disappeared (each word is uniquely represented by a trie node).

4.2 Floyd-Warshall

```
floyd_warshall :-
    % Initialize distance between all vertices to infinity
    forall((vertex(I), vertex(J)), assertz(dist(I,J,1000000))),
    % Set the distance from V to V to 0
    forall(vertex(V), set_dist(V,V,0)),
    forall(weight(U,V,W), set_dist(U,V,W)),
    forall((vertex(K), vertex(I), vertex(J)),
        (dist(I,K,D1),
         dist(K,J,D2),
         D12 is D1 + D2,
         mindist(I,J,D12))).

mindist(I,J,D) :- dist(I,J,OldD), ( D < OldD -> set_dist(I,J,D) ; true ).

set_dist(U,V,W) :- retract(dist(U,V,_)), assertz(dist(U,V,W)).
```

Fig. 6. Floyd-Warshall Code

The Floyd-Warshall algorithm computes the shortest paths problem in a weighted graph in $O(n^3)$ time, where n is the number of vertices. Let $G = (V, E)$ be a weighted directed graph, $V = v_1, \ldots, v_n$ the set of vertices, $E \subseteq V^2$, and $w_{i,j}$ the weight associated to edge (v_i, v_j) (where $w_{i,j} = \infty$ if $(v_i, v_j) \notin E$ and $w_{i,i} = 0$). The algorithm is based on incrementally updating an estimate on the shortest path between each pair of vertices until the result is optimal. Figure 6 shows a simple Prolog implementation. The code uses a dynamic predicate `dist/3` to store the computed minimal distance between each pair of vertices. For each vertex k, the distance between each (i, j) is updated with the minimum distance calculated so far.

Performance evaluation. The performance of our Floyd-Warshall implementation for different sizes of graphs is shown in Fig. 7. We consider three indexing methods for the `dist/3` predicate: *def* uses the default first order argument indexing, *t12* computes the vertex pair key using two-level indices, *p12* uses a packed integer representation (obtaining a single integer representation for the pair of vertices, which is used as key), and *p12a* combines *p12* with a specialized array to store the `dist/3` clauses. The execution times are consistent with the expected algoritmic complexity, except for *def*. The linear relative factor with the rest of methods indicates that the complexity without proper indexing is $O(n^4)$.

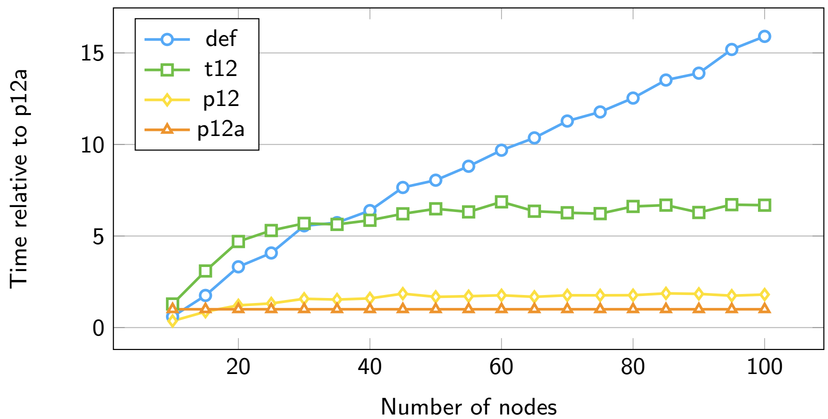

Fig. 7. Execution time for Floyd-Warshall

On the other hand, the plots also show that specialized computation of keys and data storage (*p12* and *p12a*) outperforms more generic encoding solutions (*t12*).

4.3 Module System Implementations

Module systems add the notion of modules (as separate namespaces) to predicates or terms, together with visibility and encapsulation rules. This adds a significantly complex layer on top of the program database (whether implemented in C or in Prolog meta-logic as hidden tables, as in Ciao [5]). Nevertheless, almost no changes are required in the underlying emulator machinery or program semantics. Modular terms and goals can be perfectly represented as `M:T` terms and a program transformation can systematically introduce `M` from the context. However, this would include a noticeable overhead. To solve this issue, Ciao reserves special atom names for module-qualified terms (currently, only predicates).

We can see this optimization as a particular case of pre-indexing, where the last step in module resolution (which maps to the internal representation) is a pre-indexing cast for an `mpred` *rtype*:

```
:- rtype mpred ---> nv:nv ::: index([1,0,2]).
```

For example, given a module `M = lists` and goal `G = append(X,Y,Z)`, the pre-indexed term `MG = #mpred(M:G)` can be represented as `'lists:append'(X,Y,Z)`,[8] where the first functor encodes both the module and the predicate name. To enable meta-programming, when `MG` is provided, both `M` and `G` can be recovered.

Internally, another rewrite step replaces predicate symbols by actual pointers in the bytecode, which removes yet another indirection step. This indicates that

[8] Note that the identifier does not need any symbolic description in practice.

it would be simple to reuse pre-indexing machinery for module system implementations, e.g., to enhance modules with hierarchies or provide better tools for meta-programming. In principle, pre-indexing would bring the advantages of efficient low-level code with the flexibility of Prolog-level meta representation of modules. Moreover, anchoring on `M` mimicks a memory layout where predicate tables are stored as key-value tables inside module data structures.

5 Related Work

There has been much previous work on improving indexing for Prolog and logic programming. Certain applications involving large data sets need any- and multi-argument indexing. In [3] an alternative to static generation of multi-argument indexing is presented. The approach presented uses dynamic schemes for demand-driven indexing of Prolog clauses. In [12] a new extension to Prolog indexing is proposed. User-defined indexing allows the programmer to index both instantiated and constrained variables. It is used for range queries and spatial queries, and allows orders of magnitude speedups on non-trivial datasets.

Also related is ground-hashing for tabling, studied in [15]. This technique avoids storing the same ground term more than once in the table area, based on computation of hash codes. The approach proposed adds an extra cell to every compound term to memoize the hash code and avoid the extra linear time factor.

Our work relates indexing techniques (which deal with fast lookup of terms in collections) with term representation and encoding (which clearly benefits from specialization). Both problems are related with optimal data structure implementation. Prolog code is very often used for prototyping and then translated to (low-level) imperative languages (such as C or C++) if scalability problems arise. This is however a symptom that the emulator and runtime are using suboptimal data structures which add unnecessary complexity factors. Many specialized data structures exist in the literature, with no clear winner in all cases. If they can be directly implemented in Prolog, they are often less efficient than their low-level counterparts (e.g., due to data immutability). Without proper abstraction they obscure the program to the point where a low-level implementation may not be more complex. On the other hand, adding them to the underlying Prolog machines is not trivial. Even supporting more than one term representation may have prohibitive costs (e.g., efficient implementations require a low number of tags, small code that fits in the instruction cache, etc.). Our work aims at reusing the indexing machinery when possible and specializing indexing for particular programs.

6 Conclusions and Future Work

Traditionally, Prolog systems index terms during clause selection (in the best case, reducing a linear search to constant time). Despite that, index lookup is proportional to the size of the term. In this paper we have proposed a mixed approach where indexing is precomputed during term creation. To do that, we define a notion of instantiation types and annotated constructors that specify the indexing mode. The advantage of this approach is that lookups become sub-linear. We have shown experimentally that this approach improves clause indexing and that it has other applications, for example for module system implementation.

These results suggest that it may be interesting to explore lower-level indexing primitives beyond clause indexing. This work is also connected with structure sharing. In general, pre-indexing annotations allow the optimization of simple Prolog programs with scalability problems due to data representation.

As future work, there are some open lines. First, we plan to polish the current implementation, which is mostly based on program rewriting and lacks garbage collection of indexing tables. We expect major performance gains by optimizing some operations at the WAM or C level. Second, we want to extend our repertoire of indexing methods and supporting data structures. Finally, *rtype* declarations and annotations could be discovered and introduced automatically via program analysis or profiling (with heuristics based on cost models).

References

1. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)
2. Boyer, R., More, J.: The sharing of structure in theorem-proving programs. Machine Intelligence 7 pp. 101–116 (1972)
3. Costa, V.S., Sagonas, K.F., Lopes, R.: Demand-driven indexing of prolog clauses. In: Dahl, V., Niemelä, I. (eds.) ICLP. Lecture Notes in Computer Science, vol. 4670, pp. 395–409. Springer (2007)
4. Graf, P.: Term Indexing, Lecture Notes in Computer Science, vol. 1053. Springer (1996)
5. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming 12(1–2), 219–252 (January 2012), http://arxiv.org/abs/1102.5497
6. Johnson, E., Ramakrishnan, C., Ramakrishnan, I., Rao, P.: A space efficient engine for subsumption-based tabled evaluation of logic programs. In: Middeldorp, A., Sato, T. (eds.) Functional and Logic Programming, Lecture Notes in Computer Science, vol. 1722, pp. 284–299. Springer Berlin / Heidelberg (1999)
7. Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1853–1964. Elsevier and MIT Press (2001)
8. Santos-Costa, V., Sagonas, K., Lopes, R.: Demand-Driven Indexing of Prolog Clauses . In: International Conference on Logic Programming. LNCS, vol. 4670, pp. 395–409. Springer Verlag (2007)
9. Schrijvers, T., Costa, V.S., Wielemaker, J., Demoen, B.: Towards Typed Prolog. In: Pontelli, E., de la Banda, M.M.G. (eds.) International Conference on Logic Programming. pp. 693–697. No. 5366 in LNCS, Springer Verlag (December 2008)
10. Swift, T., Warren, D.S.: Tabling with answer subsumption: Implementation, applications and performance. In: Janhunen, T., Niemelä, I. (eds.) JELIA. Lecture Notes in Computer Science, vol. 6341, pp. 300–312. Springer (2010)

11. Swift, T., Warren, D.S.: Xsb: Extending prolog with tabled logic programming. TPLP 12(1-2), 157–187 (2012)
12. Vaz, D., Costa, V.S., Ferreira, M.: User Defined Indexing. In: ICLP. pp. 372–386 (2009)
13. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025 (1983)
14. Welch, T.A.: A technique for high-performance data compression. IEEE Computer 17(6), 8–19 (1984)
15. Zhou, N.F., Have, C.T.: Efficient tabling of structured data with enhanced hash-consing. TPLP 12(4-5), 547–563 (2012)