Trie-Terms for Sets and Prolog Databases

David S. Warren

Efficient Representation of Sets of Clauses and Terms

Table of Contents

Summary

The predicates of the `prolog_db` module support an abstract datatype for sets of Prolog clauses. The underlying implementation uses a *ground* Prolog term that encodes an indexed "trie," which represents the set of clauses. The trie is built using the pre-order traversals of the heads of clauses of the set, and a variant of a radix tree at each branch point of the trie. It thus provides complete indexing (to clause heads) for terms that are bound on an initial sequence of symbols in their pre-order traversal. The representation is *canonical* in that the same set of clauses is always represented by the same trie-term, no matter the add and delete operations, and the order of their application, used to construct it.

These trie-terms represent sets of Prolog clauses (a.k.a., a Prolog Database, or PDB.) We call a set of clauses in this representation a "trie-term database." Operations are provided to assert clauses to, and retract clauses from, a given trie-term database to generate a new one reflecting the change. Also there is an operation to evaluate (or prove) a goal using the clauses of a given trie-term database. Trie-terms do not support duplicates or clause order, so cuts (and duplicate clauses) cannot be used in the clauses of trie-term databases. Trie-term databases can support Prolog meta-programming in a pure way, since trie-terms are simply Prolog terms and so earlier trie-term database states are automatically recovered on backtracking (unlike with Prolog's built-in assert and retract operations).

In fact, any Prolog term can be the head (or body) of a clause, so in particular integers, floats and variables are supported. Therefore, trie-terms can be used to represent sets of *ground* Prolog terms (i.e., ground facts), which either be interpreted simply as sets of terms or as sets of relations over Prolog terms. Operations are provided for adding elements to sets/relations generating new sets/relations, and for deletion and lookup. Also several set operations, e.g., union, intersection, and difference, are provided. These operations take advantage of the indexed structure of a trie-term to sometimes provide much better performance than more naive tuple-at-a-time algorithms. This module also provides several restricted relational operators that can also take advantage of the indexed data structure.

The complexity of adding, deleting, and looking up single elements in a trie-term is approximately linear in the the number of symbols in the clause being processed. The constant is a log factor based on the size of the embedded radix trees.

Since trie-terms are ground, they can be (and are) interned (a.k.a. hash-consed). I.e., they are copied to a global space and each distinct subterm is represented only once. With this representation, an interned trie-term is represented in Prolog as a specially named atom, which in fact encodes a pointer to its global explicit term representation. Every predicate that manipulates trie-terms converts the name to the pointer (or a pointer back to a name) whenever it uses (or generates) a trie-term. So a program that uses these trie-terms passes around "names" of the clause sets, instead of the possibly large terms representing the clause sets themselves. These interned trie-terms can be efficiently tabled since only the atomic name of a trie-term will be copied to a table. Also since these trie-terms are canonical, two trie-term names are equal if and only if the sets (trie-terms) that they name are equal. Also using answer subsumption with the subset operation on interned trie-terms may prove useful for some applications.

prolog_db

The `prolog_db` module defines a data type that supports a very general facility for manipulating sets of Prolog clauses represented as a single (interned, ground) Prolog term that can be useful for a variety of purposes. We will describe the module's functionality through a series of examples, which use progressively more complicated aspects of the data type.

Sets of Atoms (or Terms)

In its simplest form, the `prolog_db` data type can be used to manipulate sets of (ground) atoms efficiently. For many applications basic Prolog nondeterminism (perhaps with tabling) will effectively handle sets of terms, but occasionally there are applications in which a programmer wishes to maintain a term that explicitly represents a set of atoms. This is often done using a sorted list in Prolog; sorting eliminates duplicates and canonicalizes a list to a set, so then list equality implements set equality. This works well for small sets but, when sets get larger than a few 10's of items, performance can suffer signicantly. This can be avoided by using the `prolog_db` data type. The empty set is represented by the empty list; an element, say 'George', is added to it with `assert_in_db('George',[],_S1)` creating the set with 'George' added in `_S1`. For example,

```
| ?- assert_in_db('George',[],_S1),write_db(userout,_S1).
George.
```

Writing out the contents of `_S1`, we see it contains just the atom 'George'.

Sets can be compared for identity with the Prolog operation `==` (or `=`.)

```
| ?- assert_in_db('George',[],_S1),assert_in_db('Mary',_S1,_S2),
     assert_in_db('Mary',[],_S1B),assert_in_db('George',_S1B,_S2B),
     _S2 == _S2B, write_db(userout,_S2).
Mary, George.
```

Here we have added 'George' to the empty set and then added 'Mary' to that set to obtain `_S2`. Then we added 'Mary' to the empty set and added 'George' to that set to obtain `_S2B`, then checked that those two sets are equal (which they are, so it continued) and dumped the contents of one.

A set can be checked to see if it contains a constant 'George' using `in_db('George',Set1)`.

```
| ?- assert_in_db('George',[],_S1),assert_in_db('Mary',_S1,_S2),
   (in_db('George',_S2) -> writeln('contains George') ; writeln(oops)),
   (in_db('William',_S2) -> writeln(oops) ; writeln('but not William')).
contains George
but not William
```

`Set1` can be checked to see if it is a subset of `Set2` using `subset_db(Set1,Set2)`. For example,

```
| ?- assert_in_db('George',[],_S1),assert_in_db('Mary',_S1,_S2),
   (subset_db(_S1,_S2) -> writeln('correctly subset') ; writeln(oops)),
   (subset_db(_S2,[]) -> writeln('oops') ; writeln('correctly not subset')).
correctly subset
correctly not subset
```

The union of sets `Set1` and `Set2` can be computed into `Set3` using `union_db(Set1,Set2,Set3)`. For example,

```
| ?- assert_in_db('George',[],_S1),assert_in_db('Mary',[],_S2),
```

```
     union_db(_S1,_S2,_S3),dump_db(userout,_S3).
'Mary'.
'George'.
```

Other similar set operations are also provided. See the *Documentation of exports* section below for details. The time and space complexities of all these operations are at or near optimal. Note that "names" of sets, rather than the (possibly large) explicit trie-terms themselves, become the values of the Prolog set variables. This means that they can be efficiently tabled, since only the names are copied into and out of a table, rather than the explicit trie-terms themselves. To see the names of the sets, we can remove the underscores that prefix the set variable names in our previous queries. So repeating the union query above, we get:

```
| ?- assert_in_db('George',[],S1),assert_in_db('Mary',[],S2),
     union_db(S1,S2,S3),dump_db(userout,S3).
'Mary'.
'George'.

S0 = []
S1 = }]]}0000028A1F52C2A1
S2 = }]]}0000028A1F52C2C9
S3 = }]]}0000028A1F567C11
```

We see that a named set is represented as an atom whose name encodes the address of the trie-term structure in interned space. (The prefix '}]]}' is to try to avoid name conflicts with user strings.)

Of course, elements can be deleted from sets. This is done using `retractall_in_db(Atom,Set0,Set)`. For example:

```
| ?- new_db([]),assert_in_db(['George','Mary'],[],_S1),dump_db(userout,_S1),
     retractall_in_db('George',_S1,_S2),nl,dump_db(userout,_S2).
'Mary'.
'George'.

'Mary'.
```

Here we insert both 'George' and 'Mary' into the empty set (`assert_in_db/3` will accept a list in its first argument), print out that set, then remove 'George' from that set, and print out the result.

The functionality of the `prolog_db` data type that is required to efficiently support sets of atoms is the indexing it provides at a single point in a trie, and not in the multiple levels of the trie. Since for a set of atoms, there is really no trie.

Example: Paths in Graphs

We can use `prolog_db` trie-term sets to collect path edges when computing paths in a graph. Consider the following program:

```
:- import subset_db/2, assert_in_db/3, assert_in_db/2, dump_db/2 from prolog_db.

:- table path(_,_,po(subset_db(_,_))).
path(X,Y,S) :- edge(X,Y), assert_in_db((X->Y),S).
path(X,Y,S) :- path(X,Z,S1),edge(Z,Y),assert_in_db((Z->Y),S1,S).

edge(a,b).    edge(e,f).
edge(a,c).    edge(d,f).
edge(c,d).    edge(f,c).
```

```
edge(b,e).    edge(f,g).
```

Here `path(X,Y,S)` is defined for paths in the `edge/2` graph where node `Y` is reachable from node `X` along a path consisting of minimal sets of edges in set `S`. The `table` declaration uses answer subsumption to ensure that only the shortest paths (those with the minimal edge sets) are kept. So this will work for all graphs returning, for desired nodes, all minimal paths between them, e.g., those containing no loops. As an example, to find the minimal paths from `a` to `d` in the indicated graph, we can do:

```
| ?- X=a,Y=d,path(X,Y,S),nl,writeln((X-->Y)),prolog_db:dump_db(userout,S),fail.

(a --> d)
->(a,c).
->(c,d).

(a --> d)
->(a,b).
->(e,f).
->(b,e).
->(f,c).
->(c,d).

no
| ?-
```

Note that we have two such minimal paths from `a` to `d` in this graph. There are, of course, infinitely many distinct paths from `a` to `d` of the form `acd(fcd)*` in this graph.

By changing the answer subsumption aggregation predicate from `subset_db/2` to, say, `smaller_db/2` (which we could write using `compare_size_db/3`), we could compute just the shortest path(s).

Example: Too Many Parses

There are context-free grammars which for some input strings have many distinct parses, exponentially many in the length of the input string. If we are not careful, in trying to compute the parses, we may take exponential time constructing parses for substrings of the input string when the full input string is not itself in the language.

Consider the grammar:

```
a --> a, a.
a --> [a].
```

This grammar recognizes all strings consisting of one or more `a`'s. The natural way to construct parses for this grammar is with the following Prolog program:

```
:- table a/3.
a(a(P1,P2)) --> a(P1), a(P2).
a(a) --> [a].
```

in which we've added an argument to hold the Prolog term that represents the parse. The exponential behavior arises from the first rule: P1 may have `m` different values for a given substring it covers, and P2 may have `n` different values for its substring; thus a(P1,P2), the parse of the full string, would take on `m * n` different values.

We can short-circuit this combinatorial explosion by, instead of representing the parses explicitly, representing, as a trie-term, the set of parses at a position, and using that set in larger parses. We can thereby avoid the multiplication that causes the combinatorial

explosion. In this representation, in the above program `P1` and `P2` would take on as values *names* of sets of terms.

We can use answer subsumption to carry this out. Consider the following program:

```
:- import assert_in_db/3, new_db/1, is_db/1, in_db/2 from prolog_db.

:- table a(fold(collect/3,new_db/1),_,_).
a(a(P1,P2)) --> a(P1), a(P2).
a(a) --> [a].

collect(DB0,Parse,DB) :- assert_in_db(Parse,DB0,DB).

term_from_setterm(Term,ETerm) :-
    (is_db(Term)
     -> in_db(RTerm,Term),
        term_from_setterm(RTerm,ETerm)
     ; atomic(Term)
     -> ETerm = Term
     ; Term =.. [F|Args],
        term_from_setterm_list(Args,EArgs),
        ETerm =.. [F|EArgs]
    ).

term_from_setterm_list([],[]).
term_from_setterm_list([T|Ts],[ET|ETs]) :-
    term_from_setterm(T,ET),
    term_from_setterm_list(Ts,ETs).
```

The answer-subsumptive aggregation predicate is `collect/3`, which simply adds a newly computed parse to the current set. The predicate `term_from_setterm(+Term,-ETerm)` takes a term, `Term`, that may contain named trie-terms that represent sets as subcomponents, and replaced each such trie-term instance with the terms in that set, in turn. So now we can parse a string and generate its parses as follows:

```
| ?- a(_T,[a,a,a,a],[]),term_from_setterm(_T,P).
P = a(a,a(a,a(a,a)));
P = a(a,a(a(a,a),a));
P = a(a(a,a),a(a,a));
P = a(a(a,a(a,a)),a);
P = a(a(a(a,a),a),a);
no
| ?-
```

So here, the generation of the term `_T` by calling the grammar rule `a/3` takes polynomial time; it is the `term_from_setterm(_T,P)` that will take exponential time to produce the combinatorially many parses.

Sets of Relations

Another way the `prolog_db` data type can be used is to store an extensional relational database in a Prolog variable. A set of ground terms can be interpreted as defining a set of relations by interpreting the main functor symbol of each term as a relation name, and the subterms of that structure as the field values of a tuple of that relation. This is exactly how Prolog would treat such a set of terms were they to be asserted into Prolog's clause database.

For example, one can create a trie-term containing two r/2 tuples and three q/3 tuples in `_DB` with

```
| ?- assert_in_db([r(a,b),r(b,c),q(a,2,3),q(b,3,4),q(b,4,5)],_DB),
     dump_db(userout,_DB).
q(a,2,3).
q(b,4,5).
q(b,3,4).
r(a,b).
r(b,c).
```

Given a database name, we can ask queries of that database. For example,

```
| ?- assert_in_db([r(a,b),r(b,c),q(a,2,3),q(b,3,4),q(b,4,5)],_DB),
     in_db(q(b,X,Y),_DB).

X = 4
Y = 5;

X = 3
Y = 4;

no
| ?-
```

creates the desired trie database name and then evaluates the query `q(b,X,Y)` in the indicated database and nondeterministically returns the two answer pairs as shown.

We can also ask more complex queries, as seen with the following join:

```
| ?- assert_in_db([r(a,b),r(b,c),q(a,2,3),q(b,3,4),q(b,4,5)],_DB),
     in_db((r(a,Y),q(Y,Z,W)),_DB).

Y = b
Z = 4
W = 5;

Y = b
Z = 3
W = 4;

no
| ?-
```

The set operations of `union_db/3`, `intersect_db/3`, etc., can be used with sets representing relational databases. For example the union of two relational databases can be directly computed. And the operations can be very efficient. For example, if the two databases do not contain a table name in common, then their union is computed with a complexity that is dependent on the number of distinct tables but independent of their sizes.

The `prolog_db` data type provides operations designed to be used with extensional relational databases stored as `prolog_db` trie-terms. These operations take an input database, probably containing multiple relations, and return an output database that (usually) contains a new relation added by the operation. These operations depend on, and take advantage of, the structure of the trie-terms that represent sets of (tuples of) terms for their efficient implementation.

Just as the data type contains set operations on full databases, it contains set operations on tables within a single database. So, for example:

```
| ?- assert_in_db([r(a,b),r(b,c),s(1,2),s(2,3)],_DB1),
```

```
     union_in_db(r/2,s/2,t/2,_DB1,_DB), dump_db(userout,_DB).
t(1,2).
t(a,b).
t(2,3).
t(b,c).
r(a,b).
r(b,c).
s(1,2).
s(2,3).
```

initializes a `DB1` with tables `r/2` and `s/2`, then takes the two tables `r/2` and `s/2` in database `DB1` and uses `union_in_db/5` (*not* `union_db/3`) to produce a new database in `DB` that contains all tables in `DB1` plus a new table `t/2` that contains the union of tables `r/2` and `s/2`. Had `t/2` been in DB1, it would have been replaced with the new definition.

Example: Semi-Naive Transitive Closure

We'll introduce several other single database operators that are supported by the `prolog_db` abstract data type with the following more complicated example. Assume we have a trie-term relational database that contains an edge relation `e/2`. We want to compute (the converse of) the transitive closure of this `e/2` relation into a new relation `p/2`. It turns out that with the operations we have, the converse of the transitive closure is easier to compute directly, so that's what we will compute.

Consider the program:

```
:- import retractall_in_db/3, reorder_in_db/4, difference_in_db/5, is_empty_in_db/3,
    union_in_db/5, join_in_db/6, project_in_db/5 from prolog_db.
:- import assert_in_db/2, dump_db/2 from prolog_db.

test(Edges) :-
    assert_in_db(Edges,DB0),
    transclose(DB0,DB),
    dump_db(userout,DB).


transclose -->
    retractall_in_db(p(_,_)),
    reorder_in_db(e(A,B),delta(B,A)),
    transclose_loop.

transclose_loop -->
    difference_in_db(delta/2,p/2,delta/2),
    (is_empty_in_db(delta/2)
     -> []
     ;  union_in_db(delta/2,p/2,p/2),
        join_in_db(e/2,delta/2,1,delta/3),
        project_in_db(delta/3,1,delta/2),
        retractall_in_db(delta(_,_,_)),
        transclose_loop
    ).
```

The imports give us access to the `prolog_db` predicates we need. The predicate `test/1` is a simple driver that takes a list of `e/2` terms, initializes a `DB0` with them, calls `transclose/2` to compute the (converse of the) transitive closure, and then prints out the `DB` returned. The work of computing the transitive closure of `e/2` is all done in predicates `transclose/2` and `transclose_loop/2`.

Note first that we use the DCG transformation in our definitions of these two predicates, i.e., the rules are defined using the operator `-->`. The DCG transformation adds two arguments to each subgoal in the body of a DCG rule and chains those added variables. This in effect treats each subgoal in a DCG rule as a DB transformer, which takes a DB as input and produces a DB as output. This is exactly what each of these `prolog_db` predicates does, and so the DCG notation is natural for this specification.

So `transclose/2` transforms its input DB by first deleting the p/2 table (if any). It then uses `reorder_in_db/4` to generate a new table for `delta/2` consisting of the `e/2` tuples with their arguments reordered (i.e., the converse of `e/2`.) This in effect generates a new version of the `e/2` table with different indexing. Then it calls `transclose_loop/2` to iterate to compute the transitive closure of `e/2`.

The predicate `transclose_loop/2` iteratively transforms the database by accumulating the `delta/2` table into the `p/2` table and joining the `e/2` table to the new delta table at each iteration to get a new set of pairs in the transitive closure of `e/2`. So the `difference_in_db(delta/2,p/2,delta/2)` transformation removes from `delta/2` those pairs that are already in the transitive closure, since we want to continue with `delta/2` containing only new pairs. Then if there are no new pairs, the computation has converged and it returns the current DB. Note that we import `is_empty_in_db/3`, which is defined in `prolog_db` as `is_empty_in_db/2` as an identity DB transformer for use in a DCG rule. But if there are new tuples in `delta/2`, we union (accumulate) them into `p/2`. Then we perform a join of those tuples with the `e/2` tuples to create the new `delta` tuples. The `join_in_db/6` predicate takes two tables to join and performs an equi-join on them using an initial sequence of their arguments (here, just 1) and produces a table with one copy of the join argument(s) and all the other arguments of the input tables. This is a very restricted form of join and it is supported because, with this form of join, the join algorithm can take advantage of the trie-term data structure to perform efficiently. Other join forms (and sometimes even this one) can be more efficiently implemented using `in_db/2`. After the join, the `project_in_db(delta/3,1,delta/2)` projects out the first argument of the `delta/3` table generating the new `delta/2` table. Finally the `delta/3` table is deleted, and the loop is again invoked with the PDB containing the updated `p/2` table and the new `delta/2` table.

We can see the results of the computation on a given edge relation:

```
| ?- test([e(1,2),e(2,3),e(3,4),e(3,1)]).
p(4,1).
p(4,2).
p(4,3).
p(1,1).
p(1,2).
p(1,3).
p(2,1).
p(2,2).
p(2,3).
p(3,1).
p(3,2).
p(3,3).
e(1,2).
e(2,3).
e(3,4).
e(3,1).
```

Remember that `p/2` contains the *converse* of the transitive closure of `e/2`, so e.g., `p(4,1)` indicates that the pair `[1,4]` is in the transitive closure of `e/2`.

Full Prolog Databases

So far we have only stored ground terms in `prolog_db` trie-term databases. But one can add rules to these databases as well. Consider the following example:

```
| ?- assert_in_db([e(1,2),e(2,3),e(3,4),e(3,1)],DB0),
     assert_in_db([(p(X,Y) :- e(X,Y)), (p(X,Y) :- table(p(X,Z)),e(Z,Y))],DB0,DB1),
     in_db(table(p(X,Y)),DB1),writeln(p(X,Y)),fail.
p(3,3)
p(3,2)
p(3,1)
p(3,4)
p(2,2)
p(2,1)
p(2,4)
p(2,3)
p(1,1)
p(1,4)
p(1,3)
p(1,2)
```

Here we initially created a database with four `e/2` facts. Then we added two rules to it, to define `p/2` as the transitive closure of `e/2`. The recursive rule is left-recursive, so we indicate that calls to that left-recursive goal should be table-d. Then in this database with the facts and the rules, we call the goal `table(p(X,Y))`, which calls `p/2`, tabling the call. This query then returns the answers to the rule-defined `p(X,Y)`.

A few caveats: As seen in this example, tabling must be specified at the call level by wrapping the call of a goal to be tabled with the functor `table`. The trie-term databases of `prolog_db` do not support ordering or duplicates, so rules whose evalutation depend on those facilities are not supported. In particular, cuts (!) in rules are not allowed. See the detailed documentation on `in_db/2` for more particulars.

Example: STRIPS Planning

As an example of search through a graph of databases, consider a block planning problem as specified in STRIPS (ref?). The situation is a set of same-sized blocks in a configuration on a table, with some of the blocks perhaps stacked on top of others in towers. The problem is: given an initial state of the blocks configuration and a final state, what are the operations of moving the blocks that can turn the initial state into the final state? A block configuration is described by a set of facts using the predicates: `on(X,Y)` indicating block `X` is immediately on top of block `Y`; `onTable(X)` indicating that block `X` is resting directly on the table; `clear(X)` indicating that block has no block resting on top of it; `holds(X)` indicating that the hand is holding block `X`; and `handEmpty` indicating that the hand is empty. We note that `handEmpty` can be defined in terms of `holds(X)`. The hand has four actions it can perform: pickup a block from the table, put down a block on the table, stack a block on top of another block, and unstack a block from the top of another block.

Consider the following program:

```
:- import in_db/3, assert_in_db/3, retractall_in_db/3,
    dump_db/3, assert_in_db/2 from prolog_db.
```

```
pickup(X) -->
    in_db((clear(X),onTable(X),handEmpty)),
    assert_in_db([holds(X)]),
    retractall_in_db([clear(X),onTable(X)]).

putdown(X) -->
    in_db((holds(X))),
    assert_in_db([clear(X),onTable(X)]),
    retractall_in_db([holds(X)]).

stack(X,Y) -->
    in_db((holds(X),clear(Y))),
    assert_in_db([clear(X),on(X,Y)]),
    retractall_in_db([clear(Y),holds(X)]).

unstack(X,Y) -->
    in_db((clear(X),on(X,Y),handEmpty)),
    assert_in_db([clear(Y),holds(X)]),
    retractall_in_db([clear(X),on(X,Y)]).

action --> pickup(_X).
action --> putdown(_X).
action --> stack(_X,_Y).
action --> unstack(_X,_Y).

:- table action_sequence/2.
action_sequence --> action, dump_db(userout).
action_sequence --> action_sequence, action, dump_db(userout).

test(Source,Target) :-
    assert_in_db([(handEmpty :- \+ holds(_))|Source],SDB),
    assert_in_db([(handEmpty :- \+ holds(_))|Target],TDB),
    action_sequence(SDB,TDB).
```

Actions can transform the current state to a new state, but actions can be performed only if certain preconditions hold. So each action is associated with 1) a precondition, 2) a set of facts to add to the current state, and 3) a set of facts to remove from the current state. So, for example the `pickup(X)` action, as defined in this program, requires that block `X` be clear in the curent state, that it be on the table, and that the hand be empty. If these are true, then the action can be performed and a new state is generated by adding the fact that the hand holds block X, and by deleting the facts that `X` is clear, and that `X` is on the table. Similarly, the other three actions have their own preconditions, facts to add, and facts to delete to generate the state after application of the action. Again, these actions are specified as database transformers and so we use the DCG notation with its implicit database arguments.

The predicate `action_sequence/2` is defined to be the transitive closure of the action DB transformer, so it will search paths through the graph of database states whose edges are actions that transform the preceding states into the succeeding ones. We need to table this predicate since this graph clearly contains cycles. (Ref. Reza's thesis).

The predicate `test/2` sets up a desired initial state (including the rule to define `handEmpty`) and final state and calls `action_sequence/2` to carry out a sequence of actions that gets the system from the initial state to the target state. This program does not save that path, and does not search for a minimal path. These functionalities would not be difficult to add (e.g. see C.R. Ramakrishnan's paper on search.)

We give partial output for a query to find a sequence of actions to turn an initial state into a final state. We elide many intermediate states. (A total of 500 were generated.)

```
| ?- test([onTable(a),on(b,c),onTable(c),onTable(d),clear(a),clear(b),clear(d)],
    [onTable(b),on(c,b),on(a,c),on(d,a),clear(d)]).
clear(a).
clear(b).
holds(d).
on(b,c).
onTable(a).
onTable(c).
:-(handEmpty,\+(holds(A))).

clear(d).
clear(b).
holds(a).
on(b,c).
onTable(d).
onTable(c).
:-(handEmpty,\+(holds(A))).

.......

clear(a).
holds(d).
on(a,c).
on(c,b).
onTable(b).
:-(handEmpty,\+(holds(A))).

clear(d).
on(d,a).
on(a,c).
on(c,b).
onTable(b).
:-(handEmpty,\+(holds(A))).

yes
| ?-
```

Representation Details

We turn to how a database is represented as a term in the `prolog_db` data type. A `prolog_db` database is represented as a ground Prolog term (or the name of such an interned term). The term represents a trie that is indexed at each potential branch point.

The first problem is how to store a set of key-value pairs so that given a key and value, we can efficiently add the pair to the table, and given a key, efficiently access its paired value in the table. There are many data structures that support these operations. We use a variant of the radix tree data structure. We'll describe the variant we use by starting by assuming that the keys are n-bit integers. We use a 4-way tree to store the index. The last 2 bits of the key determine which of the four subtrees of the root contain the value for the key. The next 2 bits of the key determine which subtree of that subtree contains the value, etc.

So as a first approximation, if we have the following 3 6-bit integers

```
15: 00 11 11
```

```
24: 01 10 00
34: 10 00 10
```

they would be stored as in the following index tree:

```
rt( rt([],[],rt([],V24,[],[]),[]),
    []
    rt(rt([],[],V34,[]),[],[],[]),
    rt([],[],[],rt(V15,[],[],[]))
 )
```

1. The 15 (00 11 11) position is found by taking the fourth (11) subtree of the main tree, then the fourth (11) subtree of that, and then the first (00) subtree of that.
2. The 24 (or 01 10 00) position is found by taking the first (00) subtree of the main tree, and then the third subtree (10) of that, and then the second (01) subtree of that.
3. The 34 (10 00 10) position is found by taking the third (10) subtree of the main tree, then the first (00) subtree of that, and then the third (10) subtree of that.

We normally want to use a bitsize much larger than 6, maybe 32. In this case every node would be 16 hops from the root, which would incur more overhead than desired. So we shrink this data structure by (iteratively) merging a leaf node with its parent, if the parent has just one child. So the actual tree we would store for the above example is:

```
rt( rtf(V24),
    []
    rtf(V34),
    rtf(V15)
 )
```

We use the rtf-term to indicate a leaf node. A full radix tree would merge all only-children (i.e. including internal ones) with their parents. That would require a much more complex algorithm and it seems unlikely that in most of our use cases, that full collapsing would be worth the effort.

Any subtree that contains no values is represented by [ ].

The position accessed by a key integer is used to store the list of values associated with that key (using a second field of the rtf record).

We will call these trees, index trees.

So the actual details of our radix trees (index tree) is as follows:

A radix tree is: [] (if empty), or Collisions, or rt(RT1,RT2,RT3,RT4) where each RTi is a radix tree.

A Colisions structure is a term of the form: [], or rt(Key,Value,Collisions) where Key is a key, Value is its associated value, and Collisions is a Collisions structure with more Key Value pairs that collide to this location.

*/

For our trie-terms, we want to index on the symbols in a Prolog term, and not n-bit integers. But we can turn symbols into n-bit integers by choosing a hash function whose output value is an appropriately sized integer.

Now a trie-term is just a set of nested index trees, where the value associated with a symbol key is an index tree for the remainder of the trie. (The last value is the body(s) of rule(s) whose preordered head generated the symbol sequence through the trie.)

Prefix Terms

In the above section describing the `prolog_db` predicates that treat a single trie-term as representing a set of extensional relational tables, we identified the table operands by using a simple predicate/arity notation. But in fact there is a more general way to specify a set of tuples for those table operations to manipulate.

We need to look a little more closely at the trie data structure that represents sets of terms. Consider the set of terms:

```
r(a,b,c).
r(a,b,d).
r(a,e,f).
r(a,f(a,b),c).
r(b,g,c).
```

and their pre-order traversals in a trie:

```
r/3 - a - b - c
            - d
          e - f
          f/2 - a - b - c
    - b - g - c
```

The position of each symbol in the trie can be thought of as the root of a sub-trie. We can indicate such a sub-trie by using a "prefix term," which is a Prolog term whose pre-ordering consists of an initial sequence of ground symbols followed a sequence of distinct variables. For example, `r(_,_,_)` and `r(a,f(a,_),_)` are prefix terms, but `r(_,b,_)` is not. Associated with a prefix term (and with its identified sub-trie) is an integer (called the arity) that indicates the number of variables in the prefix term and thus the number of terms that would be retrieved by a pass through the determined sub-trie. The full trie always has arity 1, i.e. a full traversal through the entire trie will produce a single term. In our example, the sub-trie determined by the prefix term r(_,_,_), which is rooted after the `r/3` has arity 3, since starting at that node will retrieve 3-tuples from the sub-trie; in our example, [a,b,c], [a,b,d], [a,e,f], [a,f(a,b),c], and [b,g,c]. So the prefix term r(_,_,_) is natural way to indicate the sub-trie that determines tuples in the `r/3` table. As another example, the prefix term `r(a,f(_,_),_)` determines a sub-trie that contains triples, here just the single triple, [a,b,c].

The arguments in the call to, say, `union_in_db(P1,P2,P3,DB0,DB)` that indicate tables, i.e., `P1`, `P2`, and `P3`, can actually be prefix terms, and they refer to a set of tuples in a trie-term as described in the previous paragraph. So, in fact, a table specification of `r/3`, say, is simply an accepted short-hand for the actual prefix term `r(_,_,_)`.

Usage and interface (`prolog_db`)

- **Exports:**
  - *Predicates:*

    add/2, add/3, add_to_db/3, assert1_in_db/3, assert_in_db/2, assert_in_db/3, clause_in_db/3, clause_in_db/4, compare_size_db/3, copy_in_db/4, count_in_db/3, db_to_list/2, difference_db/3, difference_in_db/5, disjoint_db/2, disjoint_in_db/3, dump_db/2, dump_db/3, equal_in_db/3, eval_db/2, fact_in_db/2, findall_db/3, gdb/1, gfact_in_db/2, in/2, in_db/2, in_db/3, intersect_db/3, intersect_in_db/5, is_db/1, is_empty_in_db/2, is_empty_in_db/3, join_in_db/6, load_in_db/2, load_in_db/3, load_in_db/4, loadc_in_db/2, loadc_in_db/3, loadc_in_db/4, materialize_in_db/4, move_in_db/4, new_db/1, pdb_unnumbervars/4, project_in_db/5, pure_in_db/2, remove/3, reorder_in_db/4, retractall_in_db/3, sgdb/1, size_db/2, subset_db/2, subset_in_db/3, subsumed_db/2, sym_diff_db/3, sym_diff_in_db/5, union_db/3, union_in_db/5, write_db/1, write_db/2, write_label_db/2, write_label_db/3, writeln_db/1, writeln_db/2, writeln_label_db/2, writeln_label_db/3, xprod_in_db/5.
- **Other modules used:**
  - *Application modules:*

    assert, basics, consult, dcg, error_handler, gensym, machine, num_vars, setof, standard, std_xsb, string, subsumes, tables, xsb_writ.

Documentation on exports (prolog_db)

add/2: [PREDICATE]

add/2 is an alias for assert_in_db/2.

add/3: [PREDICATE]

add/3 is an alias for assert_in_db/3.

assert1_in_db/3: [PREDICATE]

assert1_in_db(+Clause,+DB0,?DB) asserts a single clause, Clause, into DB0, generating DB. Normally assert_in_db/3 should be used instead of this predicate. This predicate is needed *only* in cases in which the "clause" to be asserted is itself a list, and so interpreting it as a list of clauses would be incorrect.

assert_in_db/2: [PREDICATE]

assert_in_db(+Clause,?DB) adds a clause, Clause, (or a sequence of clauses, if Clause is a list) to the empty DB to generate DB.

assert_in_db/3: [PREDICATE]

assert_in_db(+Clause,+DB0,?DB) adds a clause, Clause, (or a sequence of clauses, if Clause is a list of clauses) to DB0 to generate DB.

`clause_in_db/3:` [PREDICATE]

`clause_in_db(?Head,?Body,+DB)` unifies, in turn, `Head` and `Body` with the head and body of rules in `DB`.

`db_to_list/2:` [PREDICATE]

`db_to_list(+DB,?List)` returns the clauses in `DB` in the list `List`, in sorted order. A fact is represented as its term; a rule is represented as a `:-/2` term.

`difference_db/3:` [PREDICATE]

`difference_db(+DB1,+DB2,?DB)` takes two `prolog_db` trie-term databases, in `DB1` and `DB2` and produces a prolog DB, in `DB`, that is the set difference of the two input databases, i.e., all clauses in `DB1` that are not in `DB2`. This is done by traversing the two input tries (and their index trees) in concert and in the process generating the output trie. This allows the output trie (and index trees) to share subterms (and sub index trees) with the input tries, and to perhaps avoid traversing large portions of the input tries. For example, if the two input databases contain the same sets of tuples, the difference immediately detects that and returns the empty database.

`disjoint_db/2:` [PREDICATE]

`disjoint_db(+DB1,+DB2)` succeeds if the prolog databases `DB1` and `DB2` are disjoint.

`dump_db/2:` [PREDICATE]

`dump_db(+Filename,+DB)` writes the clauses in DB, `DB`, to the file `Filename`. If `Filename` is 'userout', the clauses will be written to userout.

`dump_db/3:` [PREDICATE]

`dump_db(+Filename,+DB0,+DB)` is a version of `dump_db/2` that returns the input database, so that it can be used as an identity transformation in a DCG rule.

`fact_in_db/2:` [PREDICATE]

`fact_in_db(?Fact,+DB)` unifies, in turn, `Fact` with each fact in DB, `DB`. The order of returned facts is indeterminate.

`gfact_in_db/2:` [PREDICATE]

`gfact_in_db(?Term,+DB)` nondeterministically unifies `Term` with each ground fact in prolog DB `DB`.

`in/2:` [PREDICATE]

`in/2` is and alias for `in_db`.

`in_db/2:` [PREDICATE]

`in_db(+Goal,+DB)` calls the goal `Goal` in the database `DB`, binding variables in `Goal` for each instance of `Goal` that is proved using the clauses of `DB`.

`in_db/3:` [PREDICATE]

`in_db(+Goal,+DB0,-DB)` is a version of `in_db/2` that returns the input database so that it can be used as an identity transformation in a DCG rule.

`intersect_db/3:` [PREDICATE]

`intersect_db(+DB1,+DB2,?DB)` takes two prolog databases (tries), in `DB1` and `DB2`, and produces a prolog DB, in `DB`, that is the intersection of the two input databases. This is done by traversing the two input tries in concert and in the process generating the output trie. This allows the output trie to share subterms with the input tries, and to perhaps avoid traversing large portions of the input tries.

`is_db/1:` [PREDICATE]

`is_db(+DB)` succeeds if `DB` looks to have the form of a valid database reference.

`load_in_db/2:` [PREDICATE]

`load_in_db(+FileName,-DB)` reads the clauses in `FileName` and creates the database `DB` that contains them.

`load_in_db/3:` [PREDICATE]

`load_in_db(+FileName,+DB0,-DB)` reads the clauses in `FileName` and adds them to the database `DB0`, creating a new database `DB`.

`load_in_db/4:` [PREDICATE]

`load_in_db(+FileName,+ExclSet,+DB0,?DB)` reads the clauses in `FileName` and, except for those that unify with clauses in the DB `ExclSet`, asserts them into `DB0` generating `DB`. The most common use of the `ExclSet` is to avoid adding unwanted directives.

`loadc_in_db/2:` [PREDICATE]

`loadc_in_db(+FileName,-DB)` uses `read_canonical` to read the clauses in `FileName` and creates the database `DB` that contains them. The clauses in the file must be in canonical form.

`loadc_in_db/3:` [PREDICATE]

`loadc_in_db(+FileName,+DB0,-DB)` uses `read_canonical` to read the clauses in `FileName` and adds them to the database `DB0`, creating a new database `DB`. The clauses in the file must be in canonical vorm.

`loadc_in_db/4:` [PREDICATE]

`loadc_in_db(+FileName,+ExclSet,+DB0,?DB)` uses `read_canonical` to read the clauses in `FileName` and, except for those that unify with clauses in the DB `ExclSet`, asserts them into `DB0` generating `DB`. The most common use of the `ExclSet` is to avoid adding unwanted directives. The clauses in the file must be in canonical form.

`new_db/1:` [PREDICATE]

`new_db(?DB)` generates (or tests for) and empty trie-term database, `DB`.

`pure_in_db/2:` [PREDICATE]

`pure_in_db(+G,+DB)` has the same functionality as `in_db(G,DB)`, but it is implemented without cuts, so it can be used when this file is loaded into a DB by `load_in_db/2`.

`remove/3:` [PREDICATE]

`remove/3` is an alias for `retractall_in_db/3`.

`retractall_in_db/3:` [PREDICATE]

`retractall_in_db(+Goal,+DB0,?DB)` retracts all clauses whose heads unify with `Goal` (or a term in `Goal` if `Goal` is a list) from `DB0` generating `DB1`.

`size_db/2:` [PREDICATE]

`size_db(+DB,?Count)` returns the number of clauses in `DB` in `Count`.

`subset_db/2:` [PREDICATE]

`subset_db(+DB1,+DB2)` succeeds if the terms in prolog DB, `DB1`, are a subset of the terms in prolog DB, `DB2`, and fails otherwise.

`subsumed_db/2:` [PREDICATE]

`subsumed_db(+DB1,+DB2)` succeeds if database `DB1` is subsumed by database `DB2`. If the databases contain only facts, then this succeeds if the set of ground instances of `DB1` is a subset of the set of ground instances of `DB2`. If the databases contain rules, then an approximation is attempted by comparing individual rules.

`sym_diff_db/3:` [PREDICATE]

`sym_diff_db(+DB1,+DB2,?DB)` takes two databases (trie-terms), in `DB1` and `DB2` and produces a DB, in `DB`, that is the symmetric difference of the two input databases, i.e., all tuples in one but not both of `DB1` and `DB2`. This is done by traversing the two input tries in concert and in the process generating the output trie. This allows the output trie to share subterms with the input tries, and to perhaps avoid traversing large portions of the input tries.

`union_db/3:` [PREDICATE]

`union_db(+DB1,+DB2,-DB)` takes two prolog databases, in `DB1` and `DB2` and produces a prolog DB, in `DB`, that is the union of the two input databases. This is done by traversing the two input tries in concert and in the process generating the output trie. This allows the output trie to share subtries with the input tries, and to perhaps avoid traversing large portions of the input tries. For example, if the two input databases contain facts for disjoint predicates, the union just creates a new trie that points to entire definitions of the predicates in the input tries, so such a union is done in time proportional to the number of predicates, and completely independent of the size (i.e., the number of tuples) of the predicates.

`write_db/1:` [PREDICATE]

`write_db(+DB)` uses Prolog's `write/1` predicate to write to userout all the elements of prolog db `DB`. The elements are written (in no particular order) on a single lince, separated by commas and terminated by a period (full stop).

`write_db/2:` [PREDICATE]

`write_db(+OutFile,+DB)` is like `write_db(+DB)` but writes its output to the file `OutFile`

`write_label_db/2:` [PREDICATE]

`write_label_db(+Label,+DB)` first writes the term `Label` followed by a colon. Then it uses `write_db(DB)` to write out the contents of prolog db `DB`.

`write_label_db/3:` [PREDICATE]

`write_label_db(+OutFile,+Label,+DB)` is similar to `write_label_db/2` but writes it output to file `OutFile`.

`writeln_db/1:` [PREDICATE]

`writeln_db(+DB)` is like `write_db/1` but writes a newline after printing the database contents.

`writeln_db/2:` [PREDICATE]

`writeln_db(+OutFile,+DB)` writes the contents of prolog DB `DB` to file `OutFile`, followed by a newline. If `OutFile` is 'userout', then it is written to userout.

`writeln_label_db/2:` [PREDICATE]

`writeln_label_db(+Label,+DB)` is like `write_label_db/2` but writes a newline after printing the database contents.

`writeln_label_db/3:` [PREDICATE]

`writeln_label_db(+OutFile,+Label,+DB)` is like `write_label_db/3` but writes a newline after printing the database contents.

`copy_in_db/4:` [PREDICATE]

`copy_in_db(+PrefI,+Pref0,+DB0,?DB)` makes a copy of `PrefI` into `Pref0` generating `DB` from `DB0`. `PrefI` and `Pref0` must be prefix terms and have the same number of variables. (See the discussin of Prefix Terms above.) This is a very fast operation since it simply copies the pointer to the appropriate subtrie.

`difference_in_db/5:` [PREDICATE]

`difference_in_db(+P1,+P2,+P3,+DB0,?DB)` computes the set difference of two subtries in `DB0` producing a new (or replaced) subtrie in `DB`. The input tries are determined by the prefix terms of `P1` and `P2`, and the place of the resulting trie is determined by prefix term `P3`. (See `is_prefix_term/2` and `apply_op_in_db/6` for discussions of prefix terms.)

`disjoint_in_db/3:` [PREDICATE]

`disjoint_in_db(+Pref1,+Pref2,+DB)` succeeds if the tuples specified by prefix term `Pref1` are disjoint from those specified by `Pref2` in `DB`. The number of variables in `Pref1` and `Pref2` must be the same.

`equal_in_db/3:` [PREDICATE]

`equal_in_db(P1,P2,DB)` succeeds if the set of tuples specified by prefix term `P1` is the same as the set of tuples specified by prefix term `P2` in `DB`.

`intersect_in_db/5:` [PREDICATE]

`intersect_in_db(+P1,+P2,+P3,+DB0,?DB)` intersects two subtries in `DB0` producing a new (or replaced) subtrie in `DB`. The input tries are determined by the prefix terms of `P1` and `P2`, and the place of the resulting trie is determined by prefix term `P3`. (See `is_prefix_term/2` and `apply_op_in_db/6` for discussions of prefix terms.)

`is_empty_in_db/2:` [PREDICATE]

`is_empty_in_db(+Pref,+DB)` succeeds if the relation determined by prefix term `Pref` in database `DB` is empty. It fails otherwise.

`is_empty_in_db/3:` [PREDICATE]

`is_empty_in_db(+Pref,+DB0,-DB)` is a version of `is_empty_in_db/2` that returns its input database so that it can be used as an identity transformation in a DCG rule.

`join_in_db/6:` [PREDICATE]

`join_in_db(+Pref1,+Pref2,+NJoin,+Pref3,+DB0,?DB)` performs an equi-join of tuples of terms specified by prefix terms `Pref1` and `Pref2` producing a new set of terms as specified in prefix term `Pref3`. `NJoin` is the number of join variables. The join arguments are the first `NJoin` fields of the tuples specified by `Pref1` and `Pref2`. The number of variables in `Pref3` must be the sum of the numbers of variables in

`Pref1` and `Pref2` minus `NJoin`. Assuming the call to `build_db/1` builds a prolog DB in `DB0`, an example query might be:

```
| ?- build_db(DB0),join_in_db(p/3,q/4,1,r/6,DB0,DB).
```

This query would materialize the predicate r/6 into `DB`, where the r/6 definition in Prolog would be:

```
r(J,P2,P3,Q2,Q3,Q4) :- p(J,P2,P3), q(J,Q2,Q3,Q4).
```

The p/3 clauses must be ground facts. (q/4 could have rules with non-empty bodies, but I don't see any application for this.)

`materialize_in_db/4:` [PREDICATE]

`materialize_in_db(+PrefI,+Pref0,+DB0,?DB)` calls the query `PrefI` in `DB0` and asserts the term `Pref0` into `DB` for every answer. Normally every variable in `Pref0` will be in the prefix term `PrefI`.

`move_in_db/4:` [PREDICATE]

`move_in_db(+Pref1,+Pref2,+DB0,-DB)` generates `DB` that is the same as `DB0` except that the tuples of `Pref1` are copied to `Pref2` and the tuples of `Pref1` are deleted.

`project_in_db/5:` [PREDICATE]

`project_in_db(+PrefI,+NArgs,+Pref0,+DB0,?DB)` creates a new relation that projects away the first `NArgs` of the tuples of `PrefI` in `DB0`, adding the result according to `Pref0` in `DB`. `PrefI` and `Pref0` must be prefix terms. The number of variables in `PrefI` minus `NArgs` must equal the number of variables in `Pref0`. An example of its use is:

```
| ?- build_db(DB0),project_in_db(p(_,_,_,_),2,q(_,_),DB0,DB).
```

which projects away the first two arguments of the `p/4` relation in `DB0` to generate the `q/2` relation in `DB`.

`reorder_in_db/4:` [PREDICATE]

`reorder_in_db(+PrefI,+Pref0,+DB0,DB)` generates a new version of the set of tuples determined by the `PrefI` prefix term by re-ordering its arguments, putting its result in the predicate determined by `Pref0`. Note that one can use this (carefully) to change the index on `Pref1` since tuples are indexed in left-to-right order. The variables in `Pref0` must be a permutation of the variables in `PrefI`, which defines the reordering of the arguments. Note that the tries that correspond to an unchanged final sequence of variables in the two prefix terms will not be traversed in the reordering.

The following example interchanges the first two arguments of `p/4` in `DB0` to create `q/4` in `DB`:

```
| ?- build_db(DB0),reorder_in_db(p(A,B,C,D),q(B,A,C,D),DB0,DB).
```

Note that reordering can bring a later argument to the beginning to make it available for use in a call to `join_in_db/6`.

`subset_in_db/3:` [PREDICATE]

`subset_in_db(+Pref1,+Pref2,+DB)` succeeds if the tuples specified by prefix term `Pref1` are a subset of those specified by `Pref2` in `DB`. The number of variables in `Pref1` and `Pref2` must be the same.

`sym_diff_in_db/5:` [PREDICATE]

`sym_diff_in_db(+P1,+P2,+P3,+DB0,?DB)` computes the symmetric difference of two subtries in `DB0` producing a new (or replaced) subtrie in `DB`. The input tries are determined by the prefix terms of `P1` and `P2`, and the place of the resulting trie is determined by prefix term `P3`. (See `is_prefix_term/2` and `apply_op_in_db/6` for discussions of prefix terms.)

`union_in_db/5:` [PREDICATE]

`union_in_db(+P1,+P2,+P3,+DB0,?DB)` unions two subtries in `DB0` producing a new (or replaced) subtrie in `DB`. The input tries are determined by the prefix terms of `P1` and `P2`, and the place of the resulting trie is determined by prefix term `P3`. (See `is_prefix_term/2` and `apply_op_in_db/6` for discussions of prefix terms.)

`xprod_in_db/5:` [PREDICATE]

`xprod_in_db(+Pref1,+Pref2,+Pref3,+DB0,?DB)` computes the cross product of the tries determined by `Pref1` and `Pref2` and puts the result at the position determined by `Pref3`. The number of variables in `Pref3` must be the sum of the numbers of variables in `Pref1` and `Pref2`. For example, assuming the call to build_db constructs a prolog database, `DB0`, the query:

```
| ?- build_db(DB),xprod_in_db(p(_,_),q(_,_,_),r(_,_,_,_,_),DB0,DB).
```

would compute the cross product of relations p/2 and q/3 in `DB0` putting the result in r/5 (with r tuples having the p values preceding the q values). The predicate r/5 is added or replaced and the new DB `DB` is generated. (Note that, since the form p/2 is converted to prefix term p(_,_), p/2, q/3 and r/5 could be used in place of the explicit prefix terms in this query.)

The tuples of r/5 would be as though it were defined by the following Prolog rule:

```
r(P1,P2,Q1,Q2,Q3) :- r(P1,P2), q(Q1,Q2,Q3).
```

The `Pref1` values must be facts (i.e., clauses without (non-true) bodies.) The complexity of this operation is the size of the `Pref1` trie, since every leaf of the `Pref1` trie is essentially extended with a pointer to the `Pref2` trie.

`count_in_db/3:` [PREDICATE]

`count_in_db(+Pref,+DB,-Count)` returns in `Count` the number of tuples specified by prefix term `Pref` in database `DB`.

`compare_size_db/3:` [PREDICATE]

`compare_size_db(+DB1,+DB2,Order)` succeeds if DB1 has fewer elements than DB2. Often operations on two sets can be done more efficiently if one knows which set is

the smaller one. This predicate can be used to determine the smaller set. It runs (at worst) in time proportional to the size of the smaller set.

`findall_db/3:` [PREDICATE]

`findall_db(Template,Goal,DB)` is like Prolog's findall/3 except it returns the found answers in an internal prolog_db, DB, instead of in a list. It uses tabling to avoid having to build the list of (possibly duplicated) answers. DOES NOT NEST! (but should)

`gdb/1:` [PREDICATE]

`sgdb(?DB)` unifies `DB` with the global DB. If no global DB has been set (by sgdb/1), it returns an empty DB.

`sgdb/1:` [PREDICATE]

`sgdb(+DB)` makes `DB` the global prolog DB.

`clause_in_db/4:` [PREDICATE]

`clause_in_db(?Head,?Body,-Vars,+DB)` unifies, in turn, `Head` and `Body` with the head and body of rules in `DB`. `Body` contains numbervarred variables, and `Vars` is a log_list of variable values.

`pdb_unnumbervars/4:` [PREDICATE]

`pdb_unnumbervars(?TermIn,+FirstN,?TermOut,?Vars)` copies `TermIn` to `TermOut` changing all occurrences of subterms of the form '_Var'(i) to the i^th element in the log_ith list `Vars`.

`eval_db/2:` [PREDICATE]

eval_db(+PDBExpr,?PDB) evaluates an expression `PDBExpr` of PDB's and produces the resulting DB in `PDB`. The operations are: * for intersection, - for difference, + for union, and xor for symmetric difference.

`add_to_db/3:` [PREDICATE]

`add_to_db/3` is an alias for `assert1_in_db/3`.

Predicate Definition Index

(Index is nonexistent)

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, etc., are referred to in the text of the document. Note that due to limitations of the `info` format unfortunately only the first reference will appear in online versions of the document.

(Index is nonexistent)