IMPROVING CLAUSE ACCESS IN PROLOG

D. Stott Parker
Thomas W. Page, Jr.
Richard Muntz

March 1988
CSD-880024

Improving Clause Access in Prolog

D. Stott Parker

Thomas W. Page, Jr.

Richard Muntz

UCLA Computer Science Dept., Los Angeles, CA, USA 90024-1596[†]

ABSTRACT

One of the weakest aspects of Prolog is in its access to clauses. This weakness is lamentable as it makes one of Prolog's greatest strengths, its ability to treat programs as data and data as programs, difficult to exploit. This paper proposes modifications to Prolog and shows how they circumvent important problems in Prolog programming in a practical way. For example, the proposed modifications permit Prolog programs that perform efficient database query (join) processing, coroutining, and abstract machine interpretation. These modifications have been used successfully at UCLA, and should be easy to implement within any existing Prolog system.

1 Introduction

Systems that have descended from DEC-10 Prolog [6] and C-Prolog [5] contain a host of predicates for accessing clauses in the Prolog database (`assert`, `retract`, `abolish`, `clause`, `record`, `findall`, etc.). These extra-logical features have been added to Prolog out of necessity, without an accompanying theory to guide the soundness of their semantics. Consequently, it is difficult or inefficient to program tasks such as the following in Prolog:

- collect clauses of a predicate in a list (e.g., a relational database select operation)
- coroutine search through clauses of several predicates (e.g., a merge join)
- find the n^{th} clause of a predicate (e.g., for generalized clause selection with SLD resolution [3])
- write interpreters with proper meta-level clause access [7].

The next section proposes simple extensions to the Prolog *ref* mechanism and `clause` predicate. Section 3 shows how these extensions help circumvent the above difficulties in a practical way. A summary of future directions follows.

[†] This work performed under the TANGRAM project, supported by DARPA contract F29601-87-C-0072.

2 Alternative Clause Access Proposals

There are several kinds of access to clauses that we might want. The current mechanism (`clause/2`) provides *backtracking* access. *Recursive* access would be available with a list or stream interface. Zaniolo has proposed a simple cursor mechanism for Prolog that permits database-like navigational access to clauses [8]. We may also wish to view a clause as an array of terms and a predicate as an array of clauses. Each of these views requires a mechanism to name clauses, and access to the clause ordering information.

2.1 Extended Reference Mechanism

We propose extending Prolog with basic reference primitives. Every clause has associated with it a name called its *ref*. Clause ordering is made explicit by the participation of *refs* in the `first_ref` and `next_ref` relations.

`is_ref`(+Ref)	Succeeds if Ref is a clause reference, and fails otherwise.
`first_ref`(+PredName,+Arity,–Ref)	Returns Ref of the first clause for predicate PredName/Arity.
`next_ref`(+Ref,–NextRef)	Given the Ref of a clause, returns NextRef, the *ref* of the next clause. Fails if Ref currently points at the last clause of a predicate.

Table 1: Reference primitives.

2.2 Meta-circular Clause

The idealized three-line Prolog meta-interpreter

```
prove(true).
prove((A,B)) :- prove(A), prove(B).
prove(G) :- clause(G,B), prove(B).
```

illustrates the power of Prolog in describing its own basic operation. With this interpreter both the goals `prove(true)` and `prove(prove(true))` succeed. It is not well-known, however, that the goal

```
?- prove(prove(prove(true))).
```

fails with this interpreter![†] This goal fails because `clause` is not defined meta-circularly, a needless deficiency. When `clause` is repaired to include

```
clause(clause(H,B),true) :- clause(H,B).
```

[†] The interpreter also fails to respect cuts or builtin predicates, of course.

the goal succeeds as it should. That is, `clause` should be defined meta-circularly as follows:

```
clause(H,_) :- var(H), !, error('clause/2: first arg must be nonvar').
clause(clause(H,B),true) :- !, clause(H,B).
clause(H,B) :- functor(H,P,A), first_ref(P,A,R), clause_ref(H,B,R).

clause_ref(H,B,R) :- instance((H:-B),R).
clause_ref(H,B,R) :- next_ref(R,S), clause_ref(H,B,S).
```

3 Discussion

The simple additions to Prolog proposed above can facilitate the implementation of a number of important programming tasks.

3.1 Collecting Clauses

Prolog implementations of `bagof`, `setof` or `findall` typically use `assert` (or `record`) to capture bindings before they are undone by backtracking. Since `assert` is expensive, often requiring several milliseconds to execute, these predicates become quite expensive to use. This is frustrating if, for example, all we need to do is accumulate a list of the clauses of a predicate, as for computing aggregates (min, max, sum, etc.). The implementation

```
clauses(P,A,L) :- functor(H,P,A), findall((H:-B), clause(H,B), L).
```

is unnecessarily slow. The same situation arises when implementing the relational database "select" operator, which extracts all rows from a table satisfying a given predication.

Using the *ref* mechanism, we can implement `clauses` without use of `assert`, or of `findall`, `setof`, etc.:

```
clauses(P,A,[C|L]) :- first_ref(P,A,R), !, instance(C,R), clauses_after(R,L).
clauses(_,_,[]).

clauses_after(R,[C|L]) :- next_ref(R,S), !, instance(C,S), clauses_after(S,L).
clauses_after(_,[]).
```

Note that `clause` can be easily defined in terms of `clauses`:

```
clause(H,B) :- functor(H,P,A), clauses(P,A,L), member((H:-B),L).
```

3.2 Coroutining

There is no way to do real *coroutining* in most Prolog systems. Suppose that `p/2` and `q/2` are relational database predicates (tables) whose clauses are stored in sorted order. With the Prolog goal "`?- p(X,Y), q(Y,Z).`" processing of the clauses for `p/2` and `q/2` cannot be interleaved in any order but that determined by backtracking. Prolog systems supporting coroutining as an added feature have been proposed e.g. [1], but in ordinary Prolog systems the fact that backtracking is the only control mechanism for clause selection prevents interleaved execution of goals.

This shortcoming generates consternation among database workers, since it means that standard Prolog is incapable of performing *merge scans* — one of the most common efficient techniques for processing joins.

The difficulty in implementing merge joins using backtracking for control, like the difficulty in collecting clauses, is that backtracking undoes bindings. Merge joins are easy to implement using the next_ref mechanism. The two predicates can be stepped through in arbitrary order, under control of the program using `next_ref`.

3.3 Selection of the n^{th} Clause

The standard clause retrieval mechanism cannot support general SLD resolution as defined in [3]. SLD resolution rests on the use of a search rule to choose a particular clause for resolution at any given point. When done depth first, the search rule reduces to an *ordering rule*. Selection functions that operate by choosing a clause by number cannot be written in current Prolog without asserts.

The same problem is encountered, for example, when implementing the `spy` debugging predicate in Prolog. Since `spy(p/2)` requests tracing to be activated whenever `p/2` is called, a common implementation is to have `spy` modify the definition of `p/2` by adding a new first clause that calls the tracing package with itself as an argument. Unfortunately, this solution requires the tracer to ignore the first clause for `p/2`, and it is not obvious how this is best done. (Typically an `assert` is used to signal that the clause has been entered already and should subsequently fail when executed.) Selecting the n^{th} clause is easy using `next_ref`.

4 Conclusions and Future Directions

It is understandable how *refs* have come to hold a second-class status in Prolog, as they bear little connection with predicate logic. Nevertheless, it should be evident from the brief presentation here that many problems related to clause access in Prolog remain open, and a proper theory of *refs* and clause access could have important practical benefits. Note that *refs* by themselves do not require special additions to logic.

A central issue in such a theory concerns modifications. The ability to modify the `next_ref` relation directly is desirable. It would, for example, give us the ability to reorder clauses (e.g. to sort a table) without requiring wholesale retracts and asserts of the entire table. The semantics of simultaneous modification and retrieval of a predicate are not obvious [4]. Lindholm and O'Keefe suggest a semantics and an implementation via timestamps for dynamic Prolog code [2]. However, while their mechanism delivers logical semantics for clause assertion and deletion, it is not naturally applicable to clause reordering, unless reordering is viewed as retracting and asserting in order.

Like the meta-circular extension above, further extensions of `clause` would be useful for writing interpreters. One extension is to permit `clause` to execute guards of predicates itself. That is, the user-defined clause

```
p(X) :- q(X), !, r(X).
```

where `q(X)` is a guard, could be stored under `clause` essentially as

```
clause(p(X),r(X)) :- q(X), !.
```

under appropriate user control. This approach [7] enormously simplifies the writing of interpreters, since cuts then have their intended effect, and the idealized three-line interpreter may be sufficient — avoiding the need for an interpreter that respects cuts.

Another important possible extension comes from permitting clauses to be *partially ordered* by a variant of `next_ref`, rather than being totally ordered as usual. An immediate application of this extension lies in supporting deductive databases with recursive rules, a topic of great recent interest. Processing of queries that use rules such as

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

requires sophisticated analysis of the recursion, careful maintenance of temporary results of the query, and so forth.

If we permit a variant of **next_ref** to capture the partial order of ancestry among persons, this effort may be unnecessary. That is, the variant of **next_ref** can essentially store the contents of the **parent** relation, but provide very efficient access to information through use of *refs*. It seems that **next_ref** itself should not be changed to partially order the clauses of a predicate, since this change necessitates backtracking to access all of the clauses. Avoiding backtrack access was one of the reasons for introducing **next_ref** in the first place.

A final observation is that *ref*s relate directly to indexing. A very useful generalization would be to have **next_ref** comprise multiple indices for a predicate, which yield clauses in different orders, perhaps sorted on different arguments. Clearly there are many avenues for future developments here.

5 Acknowledgements

We thank Lewis Chau, Carl Kesselman, Brian Livezey, and Sanjai Narain for helpful comments.

References

[1] Clark, K.L. and F.G. McCabe, "Control facilities of IC-PROLOG," in *Expert systems in the microelectronic age*, ed. D. Michie, Edinburgh U. Press (1979).

[2] Lindholm, Timothy G. and Richard A. O'Keefe, "Efficient Implementation of a Defensible Semantics for dynamic Prolog Code," pp. 21-39 in *Logic Programming: Proceedings of the Fourth International Conference Volume 1*, ed. Jean-Louis Lassez, The MIT Press, Cambridge, Massachusetts (1987).

[3] Lloyd, J., *Principles of Logic Programming*, Springer-Verlag, New York (1984).

[4] Moss, C., "Cut and Paste — defining the impure Primitives of Prolog," pp. 686-694 in *Proc. Third International Conf. on Logic Programming, Lecture Notes in Computer Science #225*, ed. Ehud Shapiro, Springer-Verlag, London (July 14-16, 1986).

[5] Pereira, Fernando, David Warren, David Bowen, Lawrence Byrd, and Luis Pereira, *C-Prolog User's Manual Version 1.4.d.edai.*

[6] Pereira, Fernando C. N., Luis Moniz Pereira, and David H. D. Warren, "User's Guide to DECsystem-10 PROLOG," , Department of Artificial Inteligence, University of Edinburgh, Edinburgh, Scotland (September 1978).

[7] Porto, A., "Two-Level Prolog," *Proc. Conf. on Future Generation Computer Systems*, North-Holland (1984).

[8] Zaniolo, C., "Prolog — A Database Query Language for All Seasons," pp. 219-232 in *Expert Database Systems*, ed. Larry Kerschberg, Benjamin/Cummings, Menlo Park, CA (1986).