TDDD08, Tutorial 6
Writing efficient Prolog programs

Version 1.1

Who? Victor Lagerkvist, Włodek Drabent

From? Theoretical Computer Science Laboratory, Linköpings Universitet, Sweden

When? October 15, 2017

Efficient Prolog

There are many techniques that can be used to make a Prolog program more efficient. Some of them are given in this tutorial.

- Basic guidelines.
- Indexing.
- Pruning the search space
- Avoiding backtrack points
- Tail recursion.
- Difference lists etc

Basic guidelines

Correctness and completeness first.

Do not bother about efficiency unless necessary.
Use profiling.

Modern Prolog systems are good in efficiently implementing "nicely written" programs.

Complexity.

- Main aspect of efficiency:
which algorithm is being implemented, its complexity.

Worst-case running time ($O(n)$ notation) still applies for Prolog.

Basic guidelines

Order of atoms in queries.
Consider these two versions of a query:

- `| ?- student(X), father(X).`
- `| ?- father(X), student(X).`

Which is better? If we are working with a database of (1) the people in Linköping or (2) the people at LiU? In general: put the most restrictive query first.

Use the libraries

Be lazy whenever possible!

- Builtins and library methods are usually faster than hand-written code.
- `use_module(module_name)` at the prompt.
- `:- use_module(module_name)` in the code.

Lots of functions for free: `append/3`, `member/2`, `map/2`, `reverse/2` etc. See the SICStus Prolog documentation.

Unification and indexing

Unification is very fast in most Prolog systems. Failing in the unification step is faster than using a rule and then failing. Consider:

```
len3_a([_X,_Y,_Z|_Xs]).

len3_b(Xs) :- length(Xs, L), L >= 3.
```

Imagine what happens when we call `len3_a/1` or `len3_b/1` with a list of thousands of elements.

Unification and indexing

Prolog identifies rules to try based on:

- Predicate name and arity (e.g. append/3).
- The main symbol of the first argument of a procedure call.

(So called indexing on the first argument)

Place identifiers, etc, in first argument! Silly example: with the program

```
p(1, 1).
p(2, 2).
p(3, 3).
...
p(10000, 10000).
```

Queries of the form `p(999, X)` are substantially faster than queries of the form `p(X, 999)`.

SWI-Prolog – indexing on multiple arguments.

Pruning branches of the search tree

Parts of the search space (SLD-tree) may be removed by means of
once/1,
Prolog if-then-else (-> ;),
and the cut (!/1).

Backtrack points

$$H :- B_1, B_2, \ldots$$

After a success $B_1\theta_0\theta_1$ of $B_1\theta_0$,
data must be kept to facilitate backtracking to B_1:
$B_1\theta_0$, which clauses not yet used, ...

(this is called *creating a backtrack point*)

unless system determines
that there are no more answers to $B_1\theta$:
– the last matching clause for $B_1\theta$ has been used
– no backtrack points between $B_1\theta$ and $B_1\theta_0\theta_1$

Backtrack points – possible substantial memory usage.

Indexing and pruning may contribute to avoiding backtrack points.

Tail recursion

Well-known trick in functional programming. Replace recursion with iteration. Possible in Prolog when:

- The recursive call is the last one in its clause,
  $p(\ldots) :- \ldots, p(\ldots).$
- No untried alternatives for the clause
  (e.g. the clause is the last rule for the predicate).
- No backtrack points left by the clause

In such case, recursion implemented like a loop.
(No new stack frame, ...)

Consider `middle/2` from lab 2.

```
middle(X, [X]).
middle(X, [_First|Xs]) :-
    append(Middle, [_Last], Xs),
    middle(X, Middle).
```

Is `middle/2` tail-recursive?

Differences lists and open structures

We have already seen difference lists in relation to definite clause grammars.

Difference lists improve the complexity of many algorithms

– naive reverse $\rightsquigarrow$ linear reverse (a lecture)

– `quicksort` from our labs – better complexity when the result represented as a difference list