Efficient prolog?

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

Topics:

1. Basic guidelines

2. Indexing

3. Cuts, if-then-else

4. Tail recursion

5. Open structures/difference lists

See also: SICStus Prolog online manual, section "Writing Efficient Programs"

Basic guidelines

First issue: What algorithm is implemented? What is its running time?

Example: "Permutation sort" from the book.

```
sort(Input, Output) :-
    permutation(Input, Output),
    sorted(Output).
```

This program is logically correct, but very inefficient. $O(N!)$ backtracks for N elements.

Reverse

But what about this? `reverse(X, Y)`: the list `Y` is the list `X` reversed.

```
reverse([],[]).
reverse([X|Xs],Y) :-
    reverse(Xs,Ys),
    append(Ys, [X], Y).

append([],Y,Y).
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs,Ys,Zs).
```

Naïve version: $O(N)$ calls for each `append`, $O(N)$ calls to `append` by `reverse`: $O(N^2)$ calls.

Reverse, cont.

Alternative version:

```
reverse(X,Y) :- rev2(X,[],Y).

% rev2(Remaining, Reversed, Result)
rev2([],Y,Y).
rev2([X|Xs], SoFar, Y) :-
    rev2(Xs, [X|SoFar], Y).
```

The list is constructed with only one pass.

The third argument is a pure output argument: The same variable `Y` is passed through in each recursive call, and finally unified with the result in the end.

Illustration:

```
rev2([1,2,3], [], Y).  % X = 1, Xs = [2,3]
rev2([2,3], [1], Y).   % X = 2, Xs = [3]
rev2([3], [2,1], Y).   % X = 3, Xs = []
rev2([], [3,2,1], Y).
```

In the final step, `Y` is instantiated as the list `[3,2,1]`.

Basic guidelines: Ordering

The ordering of the clauses for a predicate, and the subgoals for a clause, matters. Lab 2.2 illustrates one aspect of this, but also consider these two versions of a query:

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

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

Which is better? Well, are we working with a database of everyone on campus, or everyone in Linköping?

Either way, put the most restrictive subgoal first. Also, put facts and base cases before recursive cases in a program.

Use the libraries

Be lazy: Builtins and library methods are faster than hand-written code.

`use_module(library(lists))` at the prompt (`:- use_module(etc)` in the program) gives access to `append`, `member`, etc.

See the SICStus Prolog documentation.

Let unification do the work

Failing in the unification step is faster than using a rule and encountering `N=0` later, so use specific headers. Consider:

```
len3a(L) :- length(L,N), N=3.

len3b(L) :- length(L,3).

len3c([_,_,_]).
```

Imagine a call `len3(L)` with an uninstantiated `L`. The difference should be clear.

Indexing

Prolog identifies rules to try based on:

- Predicate name and arity (e.g. `append/3`)
- Type of first argument (e.g. empty list, non-empty list, `f(_)`, symbol `anna`, number `0`...

The goal `pred(X, a)` can match anything.

The goal `pred(a,X)` can only match `pred(a, ...)`.

Place identifiers, etc, in first argument!

Indexing, cont.

Related: Use non-overlapping clauses.

Ex: Merge sort outline.

```
msort([],[]).
msort([X],[X]).
msort([X1,X2|Xs], Ys) :-
    ...split,sort,merge...
```

Effect: Only one clause ever applicable.

- No duplicate answers.
- Fewer dead-ends.
- Indexing ignores (some) irrelevant clauses.

Cuts and determinacy

There are two reasons why cuts (`!`) can increase the efficiency of a program.

- *Pruning*: Cutting off useless (possibly infinite) branches of the search tree

- *Determinacy*: Informing Prolog that no further backtracking can occur for a clause

When Prolog knows that no backtracking is possible for a predicate, it can make optimised calls (in particular, no backtracking info has to be kept in memory).

These effects can cascade (letting the caller, and the caller's caller, and its caller, make optimised calls).

Cuts and determinacy, cont.

Example: Merge sorted lists.

```
merge([], Ys, Ys).
merge(Xs, [], Xs).
merge([X|Xs], [Y|Ys], [X|Zs]) :-
    X =< Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
    X > Y, merge([X|Xs], Ys, Zs).
```

Fully functional. Improvements?

Cuts and determinacy, cont.

Improvement 1: "Non-overlapping clauses", right?

```
merge([], Ys, Ys).
merge([X|Xs], [], [X|Xs]).
merge([X|Xs], [Y|Ys], [X|Zs]) :-
    X =< Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
    X > Y, merge([X|Xs], Ys, Zs).
```

One answer for `merge([],[],Z)`. More?

Cuts and determinacy, cont.

Improvement 2: Determinacy.

```
merge([], Ys, Ys).
merge([X|Xs], [], [X|Xs]).
merge([X|Xs], [Y|Ys], [X|Zs]) :-
    X =< Y, !, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
    X > Y, merge([X|Xs], Ys, Zs).
```

Cut informs Prolog that no other rules can apply. Call to `merge` is optimised.

Cuts and determinacy, cont.

An alternative to cut: If-then-else.

`(A -> B ; C)` is an `if A then B else C` construction: `A` is called only once; if successful, call `B`, else call `C`. No backtracking into `A` occurs.

```
merge([], Ys, Ys).
merge([X|Xs], [], [X|Xs]).
merge([X|Xs], [Y|Ys], [Z|Zs]) :-
    (X =< Y ->
        Z=X,
        merge(Xs, [Y|Ys], Zs)
    ;   Z=Y,
        merge([X|Xs], Ys, Zs)
    ).
```

Cut placement

Finally, some words about cut placement.

General rule: Place the cut (if you need one) at the exact point where you know that the current path is the correct one.

```
p(X) :-
    a(X), b(X), !, c(X), d(X).
```

Here, `a(X)` and `b(X)` can be tests that are allowed to fail, but `c(X)` and `d(X)` should always succeed right away, or we may accidentally cut off a correct answer.

Remember to consider all valid calling modes of your predicate (which arguments can be instantiated or free variables?), and watch out for surprises.

Tail recursion in Prolog

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

1. The recursive call is the last subclause of its clause.

2. There are no untried alternatives for the clause (e.g. the clause is the last rule for the predicate)

3. There are no untried alternatives for the other subclauses of the clause.

In this case, Prolog does not have to perform a full recursion (create a stack frame, keep track of traceback points) but can "skip" back into the clause.

Example: `middle` from lab 2.2.

```
middle(X,[X]).
middle(X, [First|Xs]) :-
    append(Middle, [Last], Xs),
    middle(X, Middle).

append([],Y,Y).
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs,Ys,Zs).
```

Is `middle` tail recursive?

Last call of last clause. Does `append` have untried alternatives?

Obviously, if the list is unknown, but what if the list is known?

Tail recursion, cont.

Is `append` determinate with a known first argument?

```
append([],Y,Y).
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs,Ys,Zs).
```

The base case is placed before the recursion. Does that mean that the recursion is an untried alternative for `append`?

No! Indexing can tell that `append([], ...)` will not match `append([X|...],...)` (wrong type of first argument).

Conclusion: Middle is tail recursive when the list is known.

Cuts and if-then-else also have this effect.

Open structures (difference lists)

In Prolog, data structures can contain variables. This can work as data structures with holes in them (or, if you prefer, as pointers to objects to be constructed later).

Example: `t(Y,t(1,t(Y,Y)))` is a tree that contains the same unknown subtree in three places.

If `X=t(Y,t(1,Y))` and `Y` is instantiated as `t(2,2)`, then `X=t(t(2,2),t(1,t(2,2)))` from that point on.

A similar example: If `X=[1,2|Y]` and `Y` is instantiated as `[3,4]`, then `X=[1,2,3,4]` from that point on.

By instantiating `Y`, we can append data to `X` without looping over it. You only need a way to "get at" `Y`.

Difference lists

We have `X=[1,2|Y]` and want to access `Y` in order to append at the end of `X`. One way is by building a structure containing both parts: By convention, our `X` would be written `X-Y`.

This example is a difference list, a common tool in Prolog programming. `X` in the example would be written `[1,2|Y]-Y` in explicit form.

Using difference lists, we can for example implement `append(X-Xt, Y-Yt, Z-Zt)` in a single step.

The same applies to other structures where uninstantiated variables are used as pointers: Create some structure `(X,Y,Z,...)` and you can access these variables directly.

Example: `appendsingle(X-Xt,Y,Z-Zt).`

Append the element `Y` to the list `X-Xt`, producing `Z-Zt`.

```
appendsingle(X-[Y|Zt],Y,X-Zt).
```

Example query:

```
appendsingle([1,2,3|Q] - Q, 4, R).
```

We get `X = [1,2,3|Q]`, `Q = [Y|Zt]`, `Y = 4`, and

```
R = X - Zt
  = [1,2,3|Q] - Zt
  = [1,2,3|[Y|Zt]] - Zt
  = [1,2,3,Y|Zt] - Zt
  = [1,2,3,4|Zt] - Zt
```

Difference lists, cont.

Two notes about difference lists:

1. There is no new syntax involved. The notation `X-Y` is only consensus; they could equally well be written `difflist(X,Y)`.

2. They are a different structure from ordinary lists. `append` for difference lists does not work for ordinary lists.

Exercises

1. a) Write a predicate constructing an open-ended difference list from an ordinary list.

   Example: `difflist([1,2,3], [1,2,3|X]-X)`.

   b) Write `reverse` with difference lists.

2. Write a tail recursive function computing $n!$ (factorial).

   Hint: You might have to use more than two arguments.

3. Write a predicate that replaces every leaf of a binary tree, represented by `t(Left,Right)`, with the tree's smallest leaf. Your predicate should go through the tree only once. Example: `t(t(1,2),3)` becomes `t(t(1,1),1)`.

   Hint: Try using some variables as pointers.