Patterns of Efficient Prolog Programs

by Robert S. Hanmer
Lucent Technologies

Copyright ©1996 Lucent Technologies.
This paper may be reproduced for the PLOP conference and its proceedings, September, 1996. All other rights reserved.

Introduction

The patterns presented here describe the solutions to some run-time efficiency problems associated with logic programming in the Prolog language. They are used at the time of the writing of the Prolog program -- before the programs are executed. These techniques are generally Prolog implementation independent.

Efficiency for the purpose of these patterns is defined as the reduction of inference steps to achieve the first solution or speeding up the exhaustive generation of all solutions. It needs to be recognized that failing early can result in the elimination of "blind alleys", or solutions paths that ultimately will fail. Sometimes *efficiency* means making the path to the first solution as short as possible.

These small patterns work together to solve the problem of making a Prolog program efficient by providing solutions to the distinct subproblems that arise. They are presented in the pattern form so that the reader, assumed to be a relative newcomer to Prolog, can understand the reasons *why* the solution stated increases the program's efficiency.

The intents of the patterns presented here are:

- Know Your Problem: You must understand the problem and the program's goals before making changes.
- Don't Delete -- Comment!: Turn things to be deleted into comments to your program to remind you of what you did to improve efficiency.

- Duplicate Subgoal Removal: Remove any subgoals that were accidentally written multiple times.
- Ground Subgoal Deletion: Remove ground subgoals that will evaluate the same each time a rule is used.
- Precondition Addition: Check preconditions to avoid wasting effort when there are basic mismatches.
- Add a Cut: Prune dead end paths to prevent backtracking.
- Add a Cut and Delete: Eliminate subgoals that the cut forces a certain way.
- Goal Ordering: Order rules or subgoals so that the most likely comes first.

These techniques are useful in improving a program's efficiency, but this is frequently done at the expense of readability and maintainability. These are important considerations in many environments and so they need to be considered by the developer when changes are being made.

Many of these techniques are "ah-bvious" (a term used by Andrew Schnable). They can help the developer after they have written the first version of a program to identify those parts that are unnecessary and possibly hindering execution efficiency.

These patterns apply to Prolog in general, but especially to the dialect known as "C-Prolog". C-Prolog is an interpreted language. Other dialects of Prolog may suggest slightly different versions of these patterns, as well as some patterns specific to them. For example a compiled Prolog such as Quintus has further optimization based upon its use of the run-time stack. The pattern Know Your Prolog (not presented here) states that you must understand these constraints of the particular Prolog implementation that you are using.

Glossary

atom: a kind of constant. Atoms begin with lower case letters.

backtracking: when a rule fails, Prolog systems backtrack to retry previous subgoals with different variable assignments. Once all combinations in one rule have been tried, the other rules in a procedure are tried.

cut: represented as a "!", a *cut* will prevent backtracking within a rule. In effect it will lock in a solution and make the rule "deterministic".

database: the collection of all rules and facts in a program.

fact: a simple truth assertion in the database.

failure: execution that does not succeed because of some difference in facts or variable assignments.

goal: terms within a query.

procedure: the collection of all rules with the same head and the same number of arguments. The number of arguments is called the *arity.* Procedures are referred to by "procedure_name/arity".

query: asserting some subgoal and asking Prolog to try to succeed and return any instantiated variables.

rule: a logical clause of the form:

```
head :– subgoal1, subgoal2.
```

This can be read as "head if subgoal1 and subgoal2".

side-effect: a non-logical thing that a term does. Such as printing something to the screen (print/1).

subgoal: a term to the right of a ":—" in a rule.

success: execution that matches all variables and reaches the end of all subgoals in a rule.

term: predicate function that can take on either a true or false value.

variable: an element that can take on different values (can be *unified* with something else). Variables begin with capital letters.

Pattern: Know Your Problem

Problem: You think that you want to optimize your Prolog program.

Context: The program works, but is slow.

Forces: "Premature optimization is the root of all evil (or at least most of it) in programming." [Knuth, 1974]

The program might be easy to read. Changing it can hurt its readability.

The program might be convoluted because it has evolved rather than being planned.

Solution: Know your problem and the context within which your program exists. For example, know what's important: speed of execution? time to find alternate solutions? ease of making changes? Without understanding these you cannot make good choices about how to improve your program's efficiency.

Resulting Context: This pattern leads to all of the patterns that follow. Many of the techniques that they introduce will result in the loss of readability. You should know if this is acceptable before making the changes. Sometimes the readability can be preserved by commenting out subgoals, rather than by deleting them, Don't Delete--Comment!.

Pattern: Don't Delete -- Comment!

Problem: You are making a change to your program and you want to delete some part of it. How can its readability and maintainability be somewhat preserved?

Context: You are improving your program in some way that requires that it be restructured or parts be deleted.

Forces: The program is understandable (readable) now.

Any changes are likely to reduce this understandability.

To maintain your program, you must understand it.

Solution: When one of the subsequent patterns specifies that something should be deleted or reordered, make ample comments as to what you are doing and why. This might be just turning the deleted subgoals into comments (refer to the example in Add a Cut and Delete).

Comments in C-Prolog are identified by the sequence /* ... */ or by everything after a % on a line.

Resulting Context: Your program will read pretty much like it did before, but you will have incorporated the necessary efficiency changes.

Pattern: Duplicate Subgoal Removal

Problem: Once you Know Your Problem, what's the first thing to look for to make a Prolog program more efficient?

Context: A program was created and debugged but wasn't analyzed for efficiency.

Forces: Executing identical subgoals only increases program execution time.

Some identical subgoals are necessary for their side-effects. For example, having several nl/0 subgoals to print several blank lines.

Solution: When more than one occurrence of identical subgoals occurs, delete or comment out all but one of them. Goal Ordering can be used to decide which ones to delete.

Resulting Context: Each goal deleted reduces the number of inference steps by one. This pattern is closely related to Ground Subgoal Deletion.

Example: In the following rule, the second g1/0 subgoal is unnecessary and can be deleted.

```
goal :—     g1,
            g2,
            g1,             % duplicate subgoal
            write('done').
```

Pattern: Ground Subgoal Deletion

Problem: How can we efficiently deal with goals that do not have any variable references in them? These are called "ground subgoals".

Context: Sometimes a rule will check for some fact in the database or perform some function with a constant variable assignment. The subgoals in this rule are ground and will always evaluate to the same thing.

This pattern doesn't apply to ground subgoals used for their side-effects and may not apply if there are cuts in a rule.

Forces: Leaving these goals sometimes increases readability of the rule, but requires inference steps to execute.

Ground subgoals will always have the same effect upon a program's execution since there are no variables involved that might change an inference step.

Solution: Decide whether a ground subgoal will succeed or fail before execution begins. If it is true, delete it. If it is false, delete or comment out the whole rule.

Sometimes you know beforehand about the run-time environment in which a program will execute. This knowledge can be used as a part of a preprocessing phase either manually or programatically.

Generally you can move the ground goals to the left within the rule and then delete them. They can be moved to the left because either they do not contain any variable references that may alter the execution of subsequent subgoals and so could be done during preprocessing (see the pattern Goal Ordering); or they check for some fact in the database, something that can be determined prior to execution.

Resulting Context: The resulting program will execute faster because some decisions were made in preprocessing or compilation steps. Whenever a ground subgoal is deleted because it looks up a fact in the database at least one inference step can be saved in the path to the rule's first solution.

Rationale: Use whatever heuristics and prior knowledge you have to make the rule execute more efficiently. While this pattern may seem obvious, its improvement of execution may not be obvious due to the reduced readability of the rule. Don't Delete -- Comment!

Example: In the following procedure, you can delete the term man(william) if you know from analysis of the database that this will be true. If you know that man(william) will not be in your database, you can delete the entire rule, since it will undoubtedly fail.

```
grandchild(X) :—    man(william),
                    parent(william,Y),
                    parent(Y,X).
```

Pattern: Precondition Addition

Problem: How can you avoid wasting inference steps that are going to fail due to mismatch of argument types?

Context: Prolog program development where different categories of terms are being used as facts, for example words and numbers.

Forces: If you can eliminate a blind alley, the resulting program execute more efficiently.

You can't always predict what arguments are going to be passed to a rule, nor can you guarantee that they will be the correct type.

If you make preliminary inferences and then fail because of some basic flaw in the arguments, you waste extra time in backtracking.

Solution: If certain characteristics, such as whether something is or is not a number, are required for the rule to succeed, check if those characteristics are present before wasting any inference steps.

Preconditions can be classified as either syntactic or satisfaction. Syntactic checks that something is or will be of the correct type (variable, number, atom, etc.) Satisfaction preconditions check for things in range.

The pattern Goal Ordering specifies that the preconditions should be added as the first subgoals.

Resulting Context: The resulting rule will fail sooner hence the overall program will execute faster. This rule also helps make the program more resistent to data errors.

Some Prologs, especially compiled dialects, may abort execution rather than failing. This is somewhere you must Know Your Prolog. With compiled Prologs, a compile time precondition would be useful. It is important to Know Your Prolog.

Example: This example shows a simple rule without any precondition.

```
check(Y) :—   oldyear(X),
              X is Y+914.
```

When a syntactic precondition is added, we can make sure that the the variable Y is a number and so the arithmetic check will make sense.

```
check(Y) :—   number(Y),
              oldyear(X),
              X is Y+914.
```

Pattern: Add a Cut

Problem: You are developing a program that could waste a considerable amount of time backtracking to look for alternate solutions. How can you reduce the number of steps until a backtracking failure?

Context: You know *a priori* that there won't be alternate solutions. If there are mutually exclusive tests, see the pattern Add a Cut and Delete.

Forces: Occasionally we know that if a rule fails, there won't be another success found via backtracking, so the backtracking will be wasted effort.

Wasted attempts to backtrack results in the program slowing down.

Sometimes you know that you will only want the first solution to a rule.

Changing backtracking can also change the program's overall functioning.

> If you introduce cuts to obtain correct behavior when the goals are of one form, there is no guarantee that anything sensible will happen if goals of another form start appearing. [Clocksin, 1984]

Solution: Add a cut ("!") at the appropriate place to prevent backtracking.

Resulting Context: The resulting program will execute faster because it will not unnecessarily backtrack. Changing the backtracking characteristics of the program may alter its functioning, use caution when applying this pattern.

Rationale: Cuts are frequently criticized because they make programs harder to read. This needs to be weighed with the desired efficiency considerations.

The first solution to the query will not be altered by the addition of a cut.

Example: The separate/3 program separates a list of atoms and numbers into two lists, one containing atoms and the other containing the numbers. Lists are represented as [Head|Tail], where Head is the leftmost element in the list and Tail is all the remaining elements. [] is the empty list.

Old:

```
separate([X|Xs],Atoms,[X|Numbers]) :—
                    number(X),
                    separate(Xs,Atoms,Numbers).
separate([X|Xs],[X|Atoms],Numbers) :—
                    atom(X),
                    separate(Xs,Atoms,Numbers).
separate([],[],[]).
```

New:

```
separate([X|Xs],Atoms,[X|Numbers]) :—
                    number(X),
                    !,
                    separate(Xs,Atoms,Numbers).
separate([X|Xs],[X|Atoms],Numbers) :—
                    atom(X),
                    !,
                    separate(Xs,Atoms,Numbers).
separate([],[],[]).
```

In the old code if X is a number but the separate(Xs,Atoms,Numbers) fails then the next rule will check to see if X is an atom. But it can't be both, hence the second check is pointless. The cut here locks in the choice of using the first rule of separate/3.

Pattern: Add a Cut and Delete

Problem: How can you reduce the number of steps until a backtracking failure?

Context: You know *a priori* that there won't be alternate solutions. Sometimes there are two tests that are mutually exclusive that indicate which rule should be used. If there aren't mutually exclusive tests, see the pattern Add a Cut.

Forces: If the program executes both of the tests it has wasted time.

But if you don't you might have problems with backtracking and retrying a rule that is inappropriate based upon the result of the test.

Removing unnecessary parts of the rule may reduce readability of the rule.

Removing unnecessary parts of a rule results in an execution speed-up.

Cuts can change the meaning of the program by altering its backtracking behavior.

If a subgoal is being executed for its side-effects you might not be able to delete it.

Solution: Add a cut ("!") after the first test and delete the second test. The first test will either pass or fail. If it fails then the next rule must be the correct one, since the tests were mutually exclusive. If the test succeeds, then the cut after it will prevent even trying the other test, known beforehand to fail, while backtracking.

To aid readability, comment out the deleted subgoals. Don't Delete -- Comment!

Resulting Context: A program in which this pattern has been applied will not backtrack and test things that you know at development time to be true.

Rationale: This pattern essentially turns the two if-then rules into a single "meta-rule" that is an "if-then-else".

The addition of a cut is not as dangerous here as it is in Add a Cut, because of the mutually exclusive test.

Example: In this example, since the checks for less than zero and greater than or equal to zero are mutually exclusive, the second test can be removed. Incorrect execution in this case is prevented by the cut in the first rule.

Old:

```
rule(X) :—      X < 0,
                !,
                other_subgoals_1.
rule(X) :—      X >= 0,
                !,
                other_subgoals_2.
```

New:

```
rule(X) :—      X < 0,
                !,
                other_subgoals_1.
rule(X) :—      % X >= 0,       %deleted for efficiency
                % !,            %deleted for efficiency
                other_subgoals_2.
```

Pattern: Goal Ordering

Problem: What's the best way to order your procedure?

Context: You have a complex rule with many subgoals, some of which might be quite expensive to evaluate.

You also have many rules within a procedure. The ordering of the rules within a procedure isn't specified by some behavior such as recursion.

Forces: The meaning of the program might change by rearranging the parts.

Sometimes the parts can be reordered so that the most likely to succeed come first without changing the meaning of the program. Sometimes there are combinations of rules or subgoals that must be together and in a certain order.

You want to fail early so that you waste as little effort as possible in the rule execution, especially if the rule is going to be expensive in terms of inference steps.

Failing late means more backtracking to find a solution.

Sometimes some characterization of the input data indicates an efficient ordering of rules. For example, put the rules to deal with the most common form of input first.

Other capabilities of Prolog, such as recursion, may require a certain goal ordering.

Depending upon what other patterns have been applied, the meaning may be significantly altered by rearranging the goals. For example where Add a Cut and Delete has been used, the rules cannot be rearranged.

Solution: This pattern can be applied at two levels. At the individual rule level, rearrange the order of the subgoals so that they can be made to fail early. This will shorten the time to achieve failure (and by backtracking to achieve success). At the procedure level, rearrange the ordering so that the most likely things happen first.

Resulting Context: Certain other techniques, such as Precondition Addition provide explicit addition of new subgoals to cause failure early. This pattern is very dependent upon your knowing about the problem being solved by the program, refer to the pattern Know Your Problem.

Rationale: By using whatever heuristics that you know you can make the program more efficient by rearranging its parts.

Example: This is an example at the subgoal level. The grandparent/2 procedure [Sterling, 1986] can be used to find either grandparents or grandchildren depending upon which argument is a variable. If you know that you want to find grandparents given a grandchild more often than you will want to find grandchildren you would order the goals like this:

```
grandparent(GrandPar,GrandChild) :—
                parent(Y,GrandChild),
                parent(GrandPar,Y).
```

If on the other hand you will want to find grandchildren given grandparents more often, order the goals like this:

```
grandparent(GrandPar,GrandChild):—
                parent(GrandPar,Y),
                parent(Y,GrandChild).
```

Example: At the rule level, consider the separate/3 procedure introduced earlier in Add a Cut. The ordering presented earlier is preferable if the input will be primarily numbers.

The next program is equivalent, but will be faster when the input is primarily atoms.

```
separate([X|Xs],[X|Atoms],Numbers) :—
                atom(X),
                !,
                separate(Xs,Atoms,Numbers).
```

```
separate([X|Xs],Atoms,[X|Numbers]) :—
                    number(X),
                    !,
                    separate(Xs,Atoms,Numbers).
separate([],[],[]).
```

Acknowledgements

Thanks to Don Roberts for shepherding these patterns for PLOP. Also to my IH writer's workshop regulars and especially Joe Davison and Alfonso Acosta.

References

[Clocksin, 1984] Clocksin, W. F. and Mellish, C. S., Programming in Prolog, 2d Edition, Springer-Verlag, New York, 1984.

[Coelho, 1982] Coelho, H., Cotta, J. C. and Pereira, L. M., How to Solve it With Prolog, 3d Edition, Ministerio Da Habitacoa E Obras Publicas Laboratorio Nacional De Engenharia Civil, Lisbon, Portugal, 1982.

[Knuth, 1974] Knuth, D. E. "Computer Programming as an Art" in Literate Programming. CSLI, 1992.

[Pereira, 1986] Pereira, F. (ed.), C-Prolog User's Manual, Version 1.4, 1986.

[Sterling, 1986] Sterling, L. and Shapiro, E., The Art of Prolog, MIT Press, Cambridge, MA, 1986.