Logic Programming

Efficiency Issues

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University of Linz, Austria
kutsia@risc.uni-linz.ac.at

Efficiency Issues in Prolog

- Narrow the Search
- Let Unification do the Work
- Avoid `assert` and `retract`
- Understand Tokenization
- Avoid String Processing
- Recognize Tail Recursion
- Let Indexing Help
- Use Accumulators
- Use Difference Lists

Narrow the Search

Efficient programs must search efficiently.

Example

Knowledge base contains 1000 grey objects and 10 horses.

```
?- horse(X), grey(X).
```

is 100 times as fast as

```
?- grey(X), horse(X).
```

Narrow the search space as early as possible.

Narrow the Search

Example

Determine whether two lists are equal as sets.

Bad solution:

```
set_equal(L1,L2) :- permute(L1,L2).
```

N element list has $N!$ permutations.

Testing set-equality of 20-element list can require 2.4×10^{18} comparisons.

Better solution:

```
set_equal(L1,L2) :- sort(L1,L3), sort(L2,L3).
```

N-element list can be sorted in $N \log N$ steps. Faster that the first solution by a factor of more that 10^{16}.

Let Unification do the Work

Example

Write a predicate that accepts a list and succeeds if the list has three elements.

Bad solution:

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

Slightly better solution:

```
had_three_elements(L):-length(L,3).
```

Good solution:

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

Let Unification do the Work

Example

Write a predicate that accepts a list and generates from it a similar list with the first two elements swapped.

Good solution:

```
swap_first_two([A,B|Rest],[B,A|Rest]).
```

The data structures `[A,B|Rest]` and `[B,A|Rest]`, or templates for them, are created when the program is compiled, and unification gives values to the variables at run time.

Avoid `assert` and `retract`

Reasons:

- `assert` and `retract` are relatively slow and they lead to a messy logic.
- In most implementation the dynamic predicates can not run in a full compiled speed.
- The effect of `assert` and `retract` can not be undone by backtracking.
- Programs get hard to debug.

Avoid `assert` and `retract`

Legitimate Uses:

- To record new knowledge in the knowledge base.
- To store the intermediate results of a computation that *must* backtrack past the point at which it gets its result. (Think about using `setof` or `bagof` instead. It might be faster.)

Understand Tokenization

Fundamental unit: Term (numbers, atoms, structures).

- Numbers are stored in fixed-point or floating-point binary.
- Atoms are stored in a symbol table in which each atom occurs only once.
- Atoms in the program are replaced by their addresses in the symbol table (tokenization).

Understand Tokenization

- Because of tokenization the structure
  ```
  f('What a long atom this seems to be',
    'What a long atom this seems to be',
    'What a long atom this seems to be')
  ```
  is more compact than
  ```
  g(aaaaa,bbbbb,ccccc).
  ```
- To compare two atoms, even long ones, the computer needs only compare their addresses.
- By contrast, comparing lists or structures requires every element to be examined individually.

Avoid String Processing

Strings:

- Lists of numbers representing ASCII codes of characters.
- `abc` – an atom.
- `"abc"` – a list `[97, 98, 99]`.
- Strings are designed to be easily taken apart.
- Their only proper use is in situations where access to the individual characters is essential.

Recognize Tail Recursion

Recursion:

- Can be inefficient.
- Each procedure call requires information to be saved so that control can return to the calling procedure.
- If a clause calls itself 1000 times, there will be 1000 copies of its stack frame in memory.

Exception:

- Tail Recursion.
- Control need not return to the calling procedure because there is nothing more for it to do.

Recognize Tail Recursion

Tail recursion exists when:

- The recursive call is the last subgoal in the clause, and
- There are no untried alternative clauses, and
- There are no untried alternatives for any subgoal preceding the recursive call in the same clause.

Recognize Tail Recursion

Example

This predicate is tail recursive.

```
test1 :- write(hello), nl, test1.
```

Example

This predicate is not tail recursive because the recursive call is not last.

```
test2 :- test2, write(hello), nl.
```

Recognize Tail Recursion

Example

This predicate is not tail recursive because it has an untried alternative.

```
test3 :- write(hello), nl, test3.
test3 :- write(goodbye).
```

Recognize Tail Recursion

Example

This predicate is not tail recursive because a subgoal has an untried alternative.

```
test4 :- g, write(hello), nl, test4.
     g :- write(starting).
     g :- write(beginning).
```

Let Indexing Help

To match the query

```
?- f(a,b).
```

PROLOG does not look at all the clauses in the knowledge base.

It looks only the clauses for `f`.

Indexing.

Let Indexing Help

Implementation dependent.

- Many implementations index not only the predicate symbol but also the main functor of the first argument
- *First-argument indexing.*
- For `?- f(a,b).`
  The search considers only clauses that match `f(a,...)` and neglects clauses such as `f(b,c)`.

Let Indexing Help

Consequences of (first-argument) indexing.
Argument order:

- The first argument should be the one most likely to be known at search time, and
- Preferably the most diverse.
- Better to have

  ```
  f(a,x).
  f(b,x).
  f(c,x).
  ```

  than

  ```
  f(x,a).
  f(x,b).
  f(x,c).
  ```

Let Indexing Help

Consequences of (first-argument) indexing.

Indexing can make a predicate tail recursive when it otherwise would not be.

Example

```
p(f(A,B)) :- p(A).
p(a).
```

is tail-recursive because indexing eliminates `p(a)` from consideration.