

Computational Logic

Efficiency Issues in Prolog

1

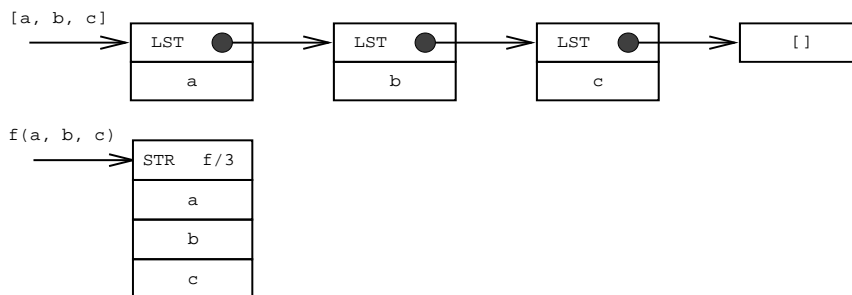
Efficiency

- In general, efficiency \equiv savings:
 - ◇ Not only time
(number of unifications, reduction steps, LIPS, etc.)
 - ◇ Also memory
- General advice:
 - ◇ Use the best algorithms
 - ◇ Use the appropriate data structures
- Each programming paradigm has its specific techniques, try not to adopt them blindly.
- The timings which will appear in the following examples have been taken on a SPARC2, under SICStus Prolog 2.1

2

Data structures

- D.H.D. Warren: “Prolog means easy pointers”
- Do not make excessive use of lists:
 - ◇ In general, only when the number of elements is unknown
 - ◇ It is convenient to keep them ordered sometimes (e.g., set equality)
 - ◇ Otherwise, use structures (functors):
 - * Less memory
 - * Direct access to each argument (arg/3) (like arrays!)



Data structures (Contd.)

- Use advanced data structures:
 - ◇ Sorted trees
 - ◇ Incomplete structures
 - ◇ Nested structures
 - ◇ ...

Let Unification Do the Work

- Unification is very powerful. Use it!
- Example: Swapping two elements of a structure:

$f(X, Y) \implies f(Y, X)$

- ◇ Slow, difficult to understand, long version:

```
swap(S1, S2):-  
    functor(S1, f, 2), functor(S2, f, 2),  
    arg(1, S1, X1), arg(2, S1, Y1),  
    arg(1, S2, X2), arg(2, S2, Y2),  
    X1 = Y2, X2 = Y1.
```

- ◇ Fast, intuitive, shorter version:

```
swap(f(X, Y), f(Y, X)).
```

Let Unification Do the Work (Contd.)

- Example: check that a list has exactly three elements.

- ◇ Weak answer:

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

(always traverses the list and computes its length)

- ◇ Better:

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

Database

- Avoid using it for simulating global variables

Example (real executions):

```
bad_count(N):-  
    assert(counting(N)),  
    even_worse.
```

```
even_worse:- retract(counting(0)).
```

```
even_worse:-  
    retract(counting(N)),  
    N > 0, N1 is N - 1,  
    assert(counting(N1)),  
    even_worse.
```

```
good_count(0).  
good_count(N):-  
    N > 0, N1 is N - 1,  
    good_count(N1).
```

bad_count(10000): 165000 bytes, 7.2 sec.

good_count(10000): 1500 bytes, 0.01 sec.

7

Database (Contd.)

- Asserting results which have been found true (lemmas).

Example (real executions):

```
fib(0, 0).  
fib(1, 1).  
fib(N, F):-  
    N > 1,  
    N1 is N - 1,  
    N2 is N1 - 1,  
    fib(N1, F1),  
    fib(N2, F2),  
    F is F1 + F2.
```

```
lfib(N, F):- lemma_fib(N, F), !.  
lfib(N, F):-  
    N > 1,  
    N1 is N - 1,  
    N2 is N1 - 1,  
    lfib(N1, F1),  
    lfib(N2, F2),  
    F is F1 + F2,  
    assert(lemma_fib(N, F)).  
:- dynamic lemma_fib/2.  
lemma_fib(0, 0). lemma_fib(1, 1).
```

fib(24, F): 4800000 bytes, 0.72 sec.

lfib(24, F): 3900 bytes, 0.02 sec. (and zero from now on)

Warning: only useful when intermediate results are reused

8

Determinism (I)

- Many problems are deterministic
- Non-determinism is
 - ◇ Useful (automatic search)
 - ◇ But expensive
- Suggestions:
 - ◇ Do not keep alternatives if they are not needed

```
member_check([X|_],X) :- !.
```

```
member_check(_|Xs,X) :- member_check(Xs,X).
```

- ◇ Program deterministic problems in a deterministic way:

Simplistic:

```
decomp(N, S1, S2):-  
    between(0, N, S1),  
    between(0, N, S2),  
    N ::= S1 + S2.
```

Better:

```
decomp(N, S1, S2):-  
    between(0, N, S1),  
    S2 is N - S1.
```

9

Determinism (II)

- Checking that two (ground) lists contain the same elements
- Naive:

```
same_elements(L1, L2):-  
    \+ (member(X, L1), \+ member(X, L2)),  
    \+ (member(X, L2), \+ member(X, L1)).
```

- 1000 elements: 7.1 secs.
- Sort and unify:

```
same_elements(L1, L2):-  
    sort(L1, Sorted),  
    sort(L2, Sorted).
```

(sorting can be done in $O(N \log N)$)

- 1000 elements: 0 secs.

10

Search order

- Golden rule: fail as early as possible (prunes branches)
- How: reorder goals in the body (perhaps even dynamically)
- Example: generate and test

```
generate_z(Z):-  
    generate_x(X),  
    generate_y(X, Y),  
    test_x(X),  
    test_y(Y),  
    combine(X, Y, Z).
```

- ◇ Perform tests as soon as possible:

```
generate_z(Z):-  
    generate_x(X),  
    test_x(X),  
    generate_y(X, Y),  
    test_y(Y),  
    combine(X, Y, Z).
```

- ◇ Even better: test as *deeply as possible* within the generator

```
generate_z(Z):-  
    generate_x_test(X),  
    generate_y_test(X, Y),  
    combine(X, Y, Z).
```

11

Indexing

- Indexing on the first argument:
 - ◇ At compile time an indexing table is built for each predicate based on the principal functor of the first argument of the clause heads
 - ◇ At run-time only the clauses with a compatible functor in the first argument are considered
- Result: appropriate clauses are reached faster and choice-points are not created if there are no “eligible” clauses left
- Improves the ability to detect determinacy, important for preserving working storage

12

Indexing (Contd.)

- Example: value greater than all elements in list

```
bad_greater(_X, []).  
bad_greater(X, [Y|Ys]):- X > Y, bad_greater(X, Ys).
```

600000 elements: 2.3 sec.

```
good_greater([], _X).  
good_greater([Y|Ys], X):- X > Y, good_greater(Ys, X).
```

600000 elements: 0.67 sec

- Can be used with structures other than lists
- Available in most Prolog systems

13

Iteration vs. Recursion

- When the recursive call is the last subgoal in the clause and there are no alternatives left in the execution of the predicate, we have an *iteration*
- Much more efficient

- Example:

```
sum([], 0).  
sum([N|Ns], Sum):-  
    sum(Ns, Inter),  
    Sum is Inter + N.
```

```
sum_iter(L, Res):-  
    sum(L, 0, Res).  
  
sum([], Res, Res).  
sum([N|Ns], In, Out):-  
    Inter is In + N,  
    sum(Ns, Inter, Out).
```

sum/2 100000 elements: 0.45 sec.

sum_iter/2 100000 elements: 0.12 sec.

14

Iteration vs. Recursion (Contd.)

- The basic skeleton is:

```
<head>:-  
    <deterministic computation>  
    <recursive_call>.
```

- Known as *tail recursion*
- Particular case of *last call optimization*
- It also consumes less memory

Cuts

- Cuts eliminate choice-points, so they “create” determinism

- Example:

```
a:-  
    test_1, !,  
    ...  
a:-  
    test_2, !,  
    ...  
...  
a:-  
    test_n, !,  
    ...
```

- If $test_1 \dots test_n$ mutually exclusive, declarative meaning of program not affected.
- Otherwise, be careful: Declarativeness, Readability.

Delaying Work

- Do not perform useless operations
- In general:
 - ◇ Do not do anything until necessary
 - ◇ Put the tests as soon as possible
- Example:
- Delaying the arithmetic operations

```
x2x3([], []).
x2x3([X|Xs], [NX|NXs]):-
    NX is -X * 2,
    X < 0,
    x2x3(Xs, NXs).
x2x3([X|Xs], [NX|NXs]):-
    NX is X * 3,
    X >= 0,
    x2x3(Xs, NXs).
```

100000 elements: 1.05 sec.

```
x2x3_1([], []).
x2x3_1([X|Xs], [NX|NXs]):-
    X < 0,
    NX is -X * 2,
    x2x3_1(Xs, NXs).
x2x3_1([X|Xs], [NX|NXs]):-
    X >= 0,
    NX is X * 3,
    x2x3_1(Xs, NXs).
```

100000 elements: 0.9 sec.

17

Delaying Work

- Delaying head unification + determinism:

```
x2x3_2([], []).
x2x3_2([X|Xs], Out):-
    X < 0, !,
    NX is -X * 2,
    Out = [NX|NXs],
    x2x3_2(Xs, NXs).
x2x3_2([X|Xs], Out):-
    X >= 0, !,
    NX is X * 3,
    Out = [NX|NXs],
    x2x3_2(Xs, NXs).
```

100000 elements: 0.68 sec. (and half the memory consumption)

- Some (personal) advice: use these techniques only when performance is essential. They might make programs:
 - ◇ Harder to understand
 - ◇ Harder to debug
 - ◇ Harder to maintain

18

Conclusions

- Avoid inheriting programming styles from other languages
- Program in a declarative way:
 - ◇ Improves readability
 - ◇ Allows compiler optimizations
- Avoid using the dynamic database when possible
- Look for deterministic computations when programming deterministic problems
- Put tests as soon as possible in the program (early pruning of the tree)
- Delay computations until needed