Concurrency

and the Logic Programming Paradigm

Joey W. Coleman, Stefan Hallerstede

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

20 May 2014

Changing the goal evaluation order

- `setX_sq(X) :- write(X), nl, X=1.`
- `?- setX_sq(X)`

Changing the goal evaluation order

- `setX_sq(X) :- write(X), nl, X=1.`
- `?- setX_sq(X)`

```
_131111
X = 1
```

Changing the goal evaluation order

- `setX_sq(X) :- write(X), nl, X=1.`
- `?- setX_sq(X)`

```
_131111
X = 1
```

- How can we produce the output `1` without changing the order of the atoms in the body of `setX_sq/1`?

Changing the goal evaluation order

- `setX_sq(X) :- write(X), nl, X=1.`
- `?- setX_sq(X)`

  ```
  _131111
  X = 1
  ```

- How can we produce the output `1` without changing the order of the atoms in the body of `setX_sq/1`?
- We have to wait until `X` is bound.

Changing the goal evaluation order

- `setX_sq(X) :- write(X), nl, X=1.`
- `?- setX_sq(X)`

  ```
  _131111
  X = 1
  ```

- How can we produce the output `1` without changing the order of the atoms in the body of `setX_sq/1`?
- We have to wait until `X` is bound.
- We say that we `freeze/2` some atoms of the body until variable `X` is bound.
- `setX_co(X) :- freeze(X,(write(X), nl)), X=1.`
- `?- setX_co(X)`

Changing the goal evaluation order

- `setX_sq(X) :- write(X), nl, X=1.`
- `?- setX_sq(X)`

```
_131111
X = 1
```

- How can we produce the output `1` without changing the order of the atoms in the body of `setX_sq/1`?
- We have to wait until `X` is bound.
- We say that we `freeze/2` some atoms of the body until variable `X` is bound.
- `setX_co(X) :- freeze(X,(write(X), nl)), X=1.`
- `?- setX_co(X)`

```
1
X = 1
```

Changing the goal evaluation order

- `setX_sq(X) :- write(X), nl, X=1.`
- `?- setX_sq(X)`

  ```
  _131111
  X = 1
  ```

- How can we produce the output `1` without changing the order of the atoms in the body of `setX_sq/1`?
- We have to wait until `X` is bound.
- We say that we `freeze/2` some atoms of the body until variable `X` is bound.
- `setX_co(X) :- freeze(X,(write(X), nl)), X=1.`
- `?- setX_co(X)`

  ```
  1
  X = 1
  ```

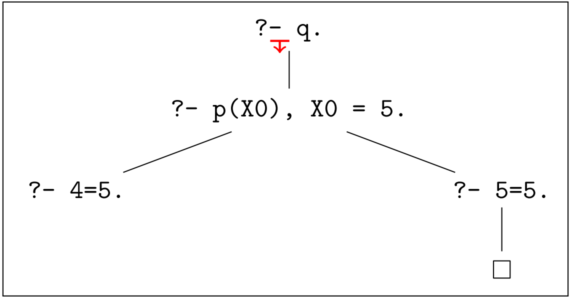
- This is the heart of Prolog's *co-routining*

Concurrency by co-routining

- Main idea:
  - *Suspend goals* when not enough information is available
  - *Resume goals* when needed information becomes available
- Search tree interpretation:
  - *Suspend branch* when not enough information is available
  - *Resume branch* when needed information becomes available
- The net effect is to reduce the size of the search tree

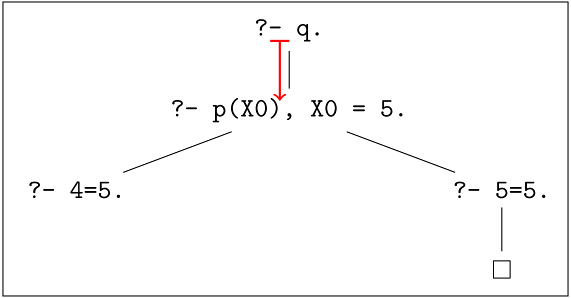
Sequential search

```
p(4).
p(5).

q :- p(X), X = 5.
```



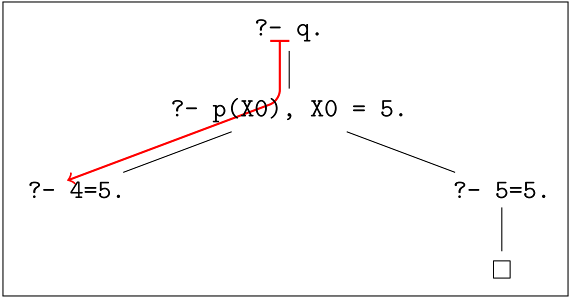
Sequential search

```
p(4).
p(5).

q :- p(X), X = 5.
```



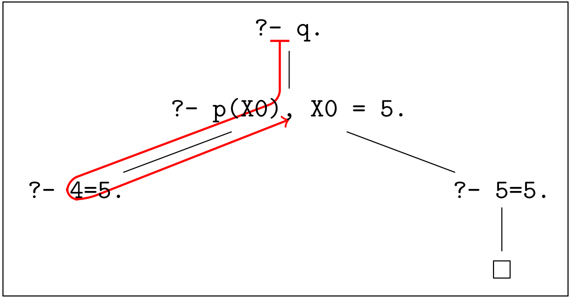
Sequential search

```
p(4).
p(5).

q :- p(X), X = 5.
```



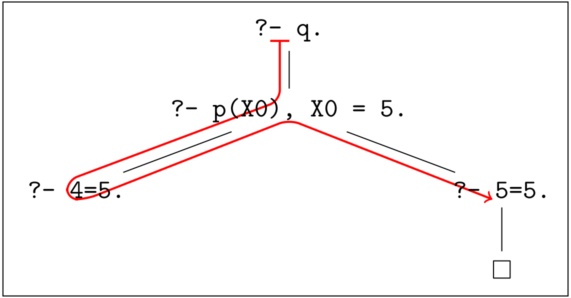
Sequential search

```
p(4).
p(5).

q :- p(X), X = 5.
```



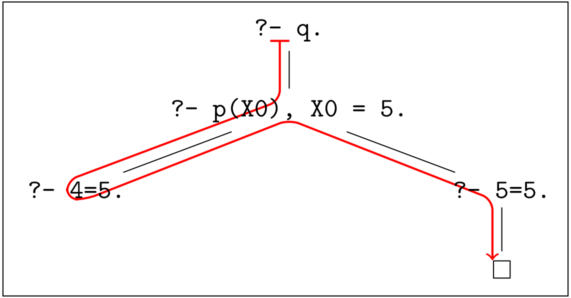
Sequential search

```
p(4).
p(5).

q :- p(X), X = 5.
```



Sequential search

```
p(4).
p(5).

q :- p(X), X = 5.
```



Co-routine search

```
p(4).
p(5).

q :- freeze(X,p(X)), X = 5.
```

```
?- q.
  |
?- freeze(X0,p(X0)), X0 = 5.
  |
?- p(5).
  |
  □
```

Co-routine search

```
p(4).
p(5).

q :- freeze(X,p(X)), X = 5.
```

```
                ?- q.
                  |
?- freeze(X0,p(X0)), X0 = 5.
                  |
              ?- p(5).
                  |
                  □
```

Co-routine search

```
p(4).
p(5).

q :- freeze(X,p(X)), X = 5.
```

```
?- q.
  |
?- freeze(X0,p(X0)), X0 = 5.
  |
?- p(5).
  |
  □
```

Co-routine search

```
p(4).
p(5).

q :- freeze(X,p(X)), X = 5.
```

```
?- q.
 |
?- freeze(X0,p(X0)), X0 = 5.
 |
?- p(5).
 |
 □
```

Co-routining basic atoms

- `freeze(X,G)` : Delay execution of goal G until the variable X is bound.
- `frozen(X,G)` : Unify `G` with a conjunction of goals suspended on variable `X`, or true if no goal has suspended.
- `when(C,G)` : Delay execution of goal `G` until the conditions `C` are satisfied.
  The conditions are of the following form:
  - `C1,C2` : Delay until both conditions `C1` and `C2` are satisfied.
  - `C1;C2` : Delay until either condition `C1` or condition `C2` is satisfied.
  - `?=(V1,V2)` : Delay until terms `V1` and `V2` have been unified.
  - `nonvar(V)` : Delay until variable `V` is bound.
  - `ground(V)` : Delay until variable `V` is ground.

  Note that `when/2` will fail if the conditions fail.
- Copied from the manuals of YAP, SWI and SICSTUS Prolog

Co-routining derived atoms

- `dif(X,Y)` : Succeed if the two arguments do not unify.
  - A call to `dif/2` will suspend if unification may still succeed or fail, and will fail if `X` and `Y` always unify.
  - If `X` and `Y` can never unify, `dif/2` succeeds deterministically.
  - If `X` and `Y` are identical it fails immediately, and finally, if `X` and `Y` can unify, goals are delayed that prevent `X` and `Y` to become equal.
  - The `dif/2` predicate behaves as if defined by

    ```
    dif(X, Y) :- when(?=(X, Y), X \== Y).
    ```

Co-routining block declarations

The declaration

```
:- block BlockSpec, ..., BlockSpec.
```

where each `BlockSpec` is a mode specification, specifies conditions for blocking goals of the predicate referred to by the mode spec (`f/3` say). When a goal for `f/3` is to be executed, the mode specs are interpreted as conditions for blocking the goal, and if at least one condition evaluates to true, the goal is blocked.

- For example, with the definition:

  ```
  :- block merge(-,?,-), merge(?,-,-).

  merge([], Y, Y).
  merge(X, [], X).
  merge([H|X], [E|Y], [H|Z]) :- H @< E,  merge(X, [E|Y], Z).
  merge([H|X], [E|Y], [E|Z]) :- H @>= E, merge([H|X], Y, Z).
  ```

  calls to `merge/3` having uninstantiated arguments in the first and third position or in the second and third position will suspend.

The behaviour of blocking goals for a given predicate on uninstantiated arguments cannot be switched off, except by abolishing or redefining the predicate.

Concurrency in Prolog

Co-routining atoms

Reasonable good hotels

Improving Generate and test

The N-queens problem

A SAT checker

Failure as negation

```
good_hotel(goedels).
good_hotel(freges).
good_hotel(tarskis).
good_hotel(quines).
expensive_hotel(goedels).
expensive_hotel(quines).

reasonable(H) :- \+expensive_hotel(H).

rg_hotel_not(H) :- reasonable(H), good_hotel(H).
```

Proper negation by co-routining

```
neg(G) :- when_bound(G, \+ G).

when_bound(Args, G) :-
        vars(Args, [], VarList),
        freeze_list(VarList, G).

freeze_list([], G) :- G.
freeze_list([Var|T], G) :- freeze(Var, freeze_list(T, G)).

vars(V, VL, VL) :- var(V), member(VV, VL), V == VV, !.
vars(V, VL, [V|VL]) :- var(V), !.
vars(V, VL, VL) :- ground(V), !.
vars([E|T], VL, VL1) :- !, vars(E, VL, VL0), vars(T, VL0, VL1).
vars(S, VL, VL1) :- compound(S), !, S =.. [_|Args], vars(Args, VL, VL1).

reasonable_neg(H) :- neg(expensive_hotel(H)).

rg_hotel_neg(H) :- reasonable_neg(H), good_hotel(H).
```

Generate and test principle

- In order to `solve` some problem we `generate` all possible candidates `X` for the solution and then `test` whether they are a solution.

  ```
  solve(X) :- generate(X), test(X)
  ```

- This can be rather inefficient.
- We would prefer to generate only as much of the candidates as necessary in case of *failures*.
- The test should be performed while generating the candidates!
- Using co-routining this can be done by blocking the test until "enough" variables have been bound and tested
- The order of the atoms `generate` and `test` is reversed:

  ```
  co_solve(X) :- test(X), generate(X)
  ```

Concurrency in Prolog

Co-routining atoms

Reasonable good hotels

Improving Generate and test

The N-queens problem

A SAT checker

N-queens sequential (Shapiro & Sterling: The Art of Prolog)

```
queens(N,Qs) :- range(1,N,Ns), permutation(Ns,Qs), safe(Qs).

range(M,N,[M|Ns]) :- M < N, M1 is M+1, range(M1,N,Ns).
range(N,N,[N]).

permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).

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

safe([Q|Qs]) :- safe(Qs), \+attack(Q,Qs,1).
safe([]).

attack(X,[Y|Ys],Z) :- X is Y+Z.
attack(X,[Y|Ys],Z) :- X is Y-Z.
attack(X,[Y|Ys],Z) :- N is Z+1, attack(X,Ys,N).
```

N-queens sequential without negation

```
queens(N,Qs) :- range(1,N,Ns), permutation(Ns,Qs), safe(Qs).

range(M,N,[M|Ns]) :- M < N, M1 is M+1, range(M1,N,Ns).
range(N,N,[N]).

permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).

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

safe([Q|Qs]) :- safe(Qs), noattack(Q,Qs,1).
safe([]).

noattack(_,[],_).
noattack(X,[Y|Ys],Z) :- Y-X=\=Z, X-Y=\=Z, N is Z + 1,
        noattack(X,Ys,N).
```

N-queens sequential without negation

```
queens(N,Qs) :- range(1,N,Ns), board(Ns,Qs), safe(Qs),
        permutation(Ns,Qs).

range(M,N,[M|Ns]) /* see above */
permutation(Xs,[Z|Zs]) /* see above */
select(X, [X|Xs], Xs) /* see above */

board([],[]).
board([N|Ns],[Q|Qs]) :- board(Ns,Qs).

safe([Q|Qs]) :- safe(Qs), noattack(Q,Qs,1).
safe([]).

noattack(_,[],_).
noattack(X,[Y|Ys],Z) :-
 when((nonvar(X),nonvar(Y)), (Y-X=\=Z, X-Y=\=Z)),
 N is Z + 1, noattack(X,Ys,N).
```

A SAT checker (Howe & King: A Pearl on SAT Solving in Prolog)

```
sat(Clauses, Vars) :-
        problem_setup(Clauses), elim_var(Vars).
elim_var([]).
elim_var([Var | Vars]) :-
        elim_var(Vars), (Var = true; Var = false).
problem_setup([]).
problem_setup([Clause | Clauses]) :-
        clause_setup(Clause),
        problem_setup(Clauses).
clause_setup([Pol-Var | Pairs]) :- set_watch(Pairs, Var, Pol).
set_watch([], Var, Pol) :- Var = Pol.
set_watch([Pol2-Var2 | Pairs], Var1, Pol1):-
        watch(Var1, Pol1, Var2, Pol2, Pairs).
:- block watch(-, ?, -, ?, ?).
watch(Var1, Pol1, Var2, Pol2, Pairs) :-
        nonvar(Var1) ->
                update_watch(Var1, Pol1, Var2, Pol2, Pairs);
                update_watch(Var2, Pol2, Var1, Pol1, Pairs).
update_watch(Var1, Pol1, Var2, Pol2, Pairs) :-
        Var1 == Pol1 -> true; set_watch(Pairs, Var2, Pol2).
```

A SAT checker (Howe & King: A Pearl on SAT Solving in Prolog)

- Input Propositional formula in CNF, e.g., $(\neg x \lor y) \land (\neg x \lor \neg z)$
- Stated in the syntax of the SAT solver:
  `[[false-X, true-Y], [false-X, false-Z]]`
- Get possible values for `X`, `Y` and `Z`:
  `sat([[false-X, true-Y], [false-X, false-Z]],[X,Y,Z])`
- What are the solutions?