Lecture 5: Prolog Programming Techniques

COMP24412: Symbolic AI

Martin Riener

School of Computer Science, University of Manchester, UK

February 2019

What happened so far

- Prolog is a Turing complete, logic based programming language
- Queries to Prolog program yields a sequence of answer substitutions
- Answers are found by backward chaining, trying the rules in order of appearance
- Suitable rules to apply are found via unification
- Functions allow the expression of arbitrary large terms, e.g. lists

Overview

1. How to write a program
2. Execution of Prolog programs
3. Beyond Datalog: Lists
4. Two simple predicates over lists
5. Termination
6. Accumulators

Outline

1. How to write a program
2. Execution of Prolog programs
3. Beyond Datalog: Lists
4. Two simple predicates over lists
5. Termination
6. Accumulators

Finding good predicate names

- Programs are written to a file (queries happen at the prompt)
- All rules defining the same predicate must appear in succession
- Programs are loaded via `consult('program.pl').`
  Careful: editors might mistake the extension for Perl

Finding good predicate names

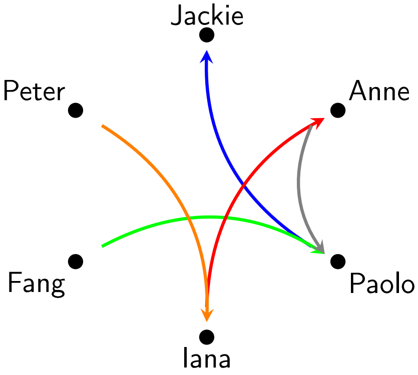
- Programs are written to a file (queries happen at the prompt)
- All rules defining the same predicate must appear in succession
- Programs are loaded via `consult('program.pl')`.
  Careful: editors might mistake the extension for Perl

- Predicates describe relations, don't assume a direction of evaluation:
  Compare `find(car, List)` to `member_of(car, List)`
- Use short phrases for each argument, use _ to seperate them:
  Compare `flight(X,Y,Z)` to `flightno_from_to(X,Y,Z)`

Recursively defined predicates: inductive reasoning

- Formulate simplest facts
- Find rules that extend smaller terms to (slightly) larger terms

A ball game



Representing the graph

```
child_throwsto(peter, iana).
child_throwsto(iana, anne).
child_throwsto(fang, paolo).
child_throwsto(paolo, jackie).
child_throwsto(anne, paolo).
```

Tossing the ball around

```
ballfrom_reaches(From,To) :-
    child_throwsto(From,To).
```

Tossing the ball around

```
ballfrom_reaches(From,To) :-
    child_throwsto(From,To).
ballfrom_reaches(From,To) :-
    child_throwsto(From,Neighbour),
    ballfrom_reaches(Neighbour,To).
```

Tossing the ball around

Why not?

```
ballfrom_reaches(From,To) :-
    ballfrom_reaches(From,Neighbour),
    ballfrom_reaches(Neighbour,To).
```

- No constraint on the solution in the first recursion step
- Leads to an infinite recursion

Outline

1. How to write a program
2. Execution of Prolog programs
3. Beyond Datalog: Lists
4. Two simple predicates over lists
5. Termination
6. Accumulators

Prolog's execution mechanism

- Prolog applies backwards reasoning (start with the query)
- Multiple goals are derived left-to-right
- Search for unifiers with rule heads, top-to-bottom
- Head unifies:
  - try to derive the goals of the rule body
  - continue fulfilling the original goal
- No head unifies: backtrack and try next rule!

A simple query

```
?- ballfrom_reaches(iana,paolo).
```

A simple query

```
?- ballfrom_reaches(iana,paolo).
true
```

A simple query

```
?- ballfrom_reaches(iana,paolo).
true ;
false.
```

A simple query

```
?- ballfrom_reaches(iana,paolo).
true ;
false.
```

What is happening?

Step-by-step execution

goal(s)

```
ballfrom_reaches(iana,paolo)
```

trying rule

```
ballfrom_reaches(From,To) :-
     child_throwsto(From,To).
```

Step-by-step execution

goal(s)

`ballfrom_reaches(iana,paolo)`

trying rule

```
ballfrom_reaches(From,To) :-
    child_throwsto(From,To).
```

instance: `From=iana, To=paolo`

```
ballfrom_reaches(iana,paolo) :-
    child_throwsto(iana,paolo).
```

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`

trying rule

```
child_throwsto(peter, iana).
```

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`

trying rule

`child_throwsto(peter, iana).`

instance: `iana` $\neq$ `peter` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`

trying rule

```
child_throwsto(iana, anne).
```

Step-by-step execution

goal(s)

`child_throwsto(iana,paolo)`

trying rule

`child_throwsto(iana, anne).`

instance: `paolo` $\neq$ `anne` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`

trying rule

`child_throwsto(fang, paolo).`

instance: `iana` $\neq$ `fang` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`

trying rule

`child_throwsto(paolo, jackie).`

instance: `iana` $\neq$ `paolo` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`

trying rule

`child_throwsto(anne, paolo).`

instance: `iana` $\neq$ `anne` – backtrack!

no more `child_throwstos` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`

trying rule

```
child_throwsto(anne, paolo).
```

Step-by-step execution

goal(s)
`ballfrom_reaches(iana,paolo)`

trying rule

```
ballfrom_reaches(From,To) :-
     child_throwsto(From,Neighbour),
     ballfrom_reaches(Neighbour,To).
```

Step-by-step execution

goal(s)

`ballfrom_reaches(iana,paolo)`

trying rule

```
ballfrom_reaches(From,To) :-
     child_throwsto(From,Neighbour),
     ballfrom_reaches(Neighbour,To).
```

instance: From=iana, To=paolo

```
ballfrom_reaches(iana,paolo) :-
     child_throwsto(iana,Neighbour),
     ballfrom_reaches(Neighbour,paolo).
```

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),` `ballfrom_reaches(Neighbour,paolo)`

trying rule

```
child_throwsto(peter, iana).
```

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),`
`ballfrom_reaches(Neighbour,paolo)`

trying rule

instance: `iana` $\neq$ `peter` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),` `ballfrom_reaches(Neighbour,paolo)`
trying rule `child_throwsto(iana, anne).` instance: Neighbour=anne – next goal!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`

trying rule

`child_throwsto(peter, iana).`

instance: `anne` $\neq$ `peter` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`

trying rule

`child_throwsto(iana, anne).`

instance: `anne` $\neq$ `iana` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`

trying rule

`child_throwsto(fang, paolo).`

instance: `anne` $\neq$ `fang` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`

trying rule

`child_throwsto(paolo, jackie).`

instance: `anne` $\neq$ `paolo` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`

trying rule

```
child_throwsto(anne, paolo).
```

instance: no substitution needed

Step-by-step execution

goal(s)

---

no goals! tell the user!

Step-by-step execution

goal(s)
user wants more solutions, backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`

trying rule

no more `child_throwstos` – backtrack!

Step-by-step execution

goal(s)

`child_throwsto(iana,Neighbour),`
`ballfrom_reaches(Neighbour,paolo)`

trying rule

`child_throwsto(fang, paolo).`

instance: `iana` $\neq$ `fang` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),`   `ballfrom_reaches(Neighbour,paolo)`
trying rule   instance: `iana` $\neq$ `paolo` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),` `ballfrom_reaches(Neighbour,paolo)`

trying rule

instance: `iana` $\neq$ `anne` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),` `ballfrom_reaches(Neighbour,paolo)`
trying rule
etc. etc. etc.

Step-by-step execution

goal(s)

---

all paths exhausted, report false!

Outline

1. How to write a program
2. Execution of Prolog programs
3. Beyond Datalog: Lists
4. Two simple predicates over lists
5. Termination
6. Accumulators

Can we make the steps visible?

- Introduce an additional argument:

```
ballfrom_reaches_via(From,To,direct(To)) :-
    child_throwsto(From,To).
ballfrom_reaches_via(From,To,next(Neighbour,Others)) :-
    child_throwsto(From,Neighbour),
    ballfrom_reaches_via(Neighbour,To,Others).
```

Can we make the steps visible?

- Query:

```
?- ballfrom_reaches_via(iana,paolo, Path).
```

Can we make the steps visible?

- Query:

```
?- ballfrom_reaches_via(iana,paolo, Path).
Path = next(anne, direct(paolo)) ;
false.
```

Can we make the steps visible?

- Query:

```
?- ballfrom_reaches_via(From,To, next(A,next(B,direct(C)))).
```

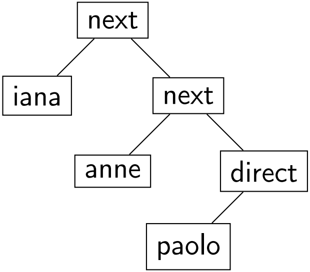
Can we make the steps visible?

- Query:

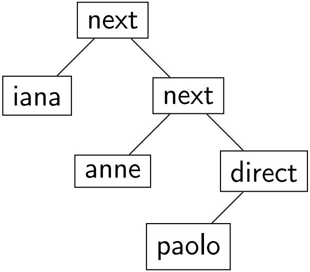
```
?- ballfrom_reaches_via(From,To, next(A,next(B,direct(C)))).
From = peter,
To = C, C = paolo,
A = iana,
B = anne ;
From = iana,
To = C, C = jackie,
A = anne,
B = paolo ;
false.
```

Data-structures: Lists

- We used `next/2` and `direct/1` to track paths
- Term graph of `next(iana, next(anne,direct(paolo)))`:



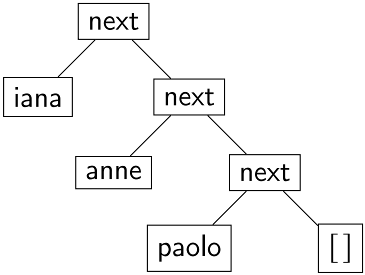
- We used `next/2` and `direct/1` to track paths
- Term graph of `next(iana, next(anne,direct(paolo)))`:



- What about an empty path?

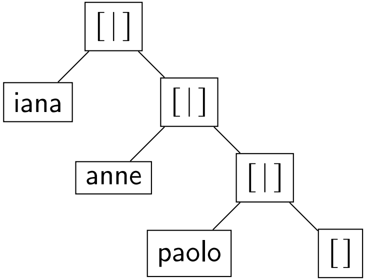
Data-structures: Lists

- Replace `direct/1` with `[]/0`
- Term graph of `next(iana, next(anne,next(paolo,[])))` :



Data-structures: Lists

- Rename next/2 to [|]/2,
- Term graph of `[iana | [anne | [paolo | [] ]]]` :



Data-structures: Lists

- Structure built over []/0 and [|]/2: linked list
- Datatype:
  - "The empty list is a list"
    ```
    isa_list([]).
    ```
  - "Any head prepended to a tail list is a list"
    ```
    isa_list([Head | Tail]) :-
          isa_list(Tail).
    ```
- Special list notation:
  - No need to append to empty list:
    `[Head | []] = [Head]`
  - Use , to avoid writing the tail in squarebrackets:
    `[X | [Y | Tail] ] = [X, Y | Tail]`

Properties of linked lists

- Access to head: $O(1)$
  `List = [Head | _ ]`
- Access to tail: $O(1)$
  `List = [_ | Tail ]`
- Traversal: $O(n)$
  `member_of(X,List)`

Outline

1. How to write a program
2. Execution of Prolog programs
3. Beyond Datalog: Lists
4. Two simple predicates over lists
5. Termination
6. Accumulators

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.
- Find base case(s):

```
member_of(X,[Head|_Tail]) :- % X is member of a list
      X = Head.              % if X is the head of the list
```

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.
- Find base case(s):

```
member_of(X,[X|_Tail]). % directly unify in head
```

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.
- Find base case(s):

```
member_of(X,[X|_Tail]). % directly unify in head
```

- Find recursive case(s):
  "Given a smaller term, how to extend it to a larger one?"

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.
- Find base case(s):

```
member_of(X,[X|_Tail]). % directly unify in head
```

- Find recursive case(s):
  "Given a smaller term, how to extend it to a larger one?"

```
member_of(X,[_Head|Tail]) :- % X is member of the list
    member_of(X, Tail).      % if X is member of the tail
```

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate `nonmember_of(X,List)` that is true whenever X is *not* an element of list $List$.

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate nonmember_of(X,List) that is true whenever X is *not* an element of list $List$.
- Find base case(s):

```
nonmember_of(_X,[]).  % Any element is not in the empty list
```

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate nonmember_of(X,List) that is true whenever X is *not* an element of list $List$.
- Find base case(s):

```
nonmember_of(_X,[]).  % Any element is not in the empty list
```

- Find recursive case(s):

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate `nonmember_of(X,List)` that is true whenever X is *not* an element of list $List$.
- Find base case(s):

```
nonmember_of(_X,[]).  % Any element is not in the empty list
```

- Find recursive case(s):

```
nonmember_of(X,[Head|Tail]) :-
    % fill in constraint
    nonmember_of(X, Tail). % X is not in the tail
```

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate `nonmember_of(X,List)` that is true whenever X is *not* an element of list $List$.
- Find base case(s):

```
nonmember_of(_X,[]).  % Any element is not in the empty list
```

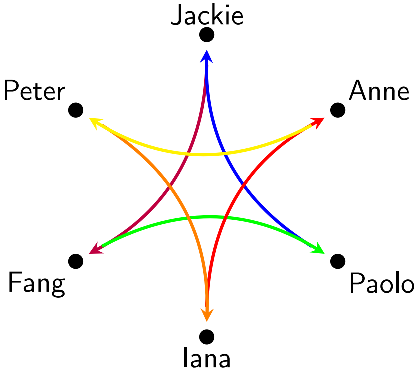
- Find recursive case(s):

```
nonmember_of(X,[Head|Tail]) :-
    dif(X,Head),           % X is different from the head
    nonmember_of(X, Tail). % X is not in the tail
```

Outline

1. How to write a program
2. Execution of Prolog programs
3. Beyond Datalog: Lists
4. Two simple predicates over lists
5. Termination
6. Accumulators

More ball games



Representing the graph

```
child_throwsto(anne, peter).
child_throwsto(peter, iana).
child_throwsto(iana, anne).
child_throwsto(jackie, fang).
child_throwsto(fang, paolo).
child_throwsto(paolo, jackie).
```

Tossing the ball around

```
ballfrom_reaches_lvia(From,To,[To]) :-
    child_throwsto(From,To).
ballfrom_reaches_lvia(From,To,[Neighbour|Path]) :-
    child_throwsto(From,Neighbour),
    ballfrom_reaches_lvia(Neighbour,To,Path).
```

Some Queries

```
?- ballfrom_reaches_lvia(jackie, paolo,Path).
Path = [fang, paolo] ;
Path = [fang, paolo, jackie, fang, paolo] ;
Path = [fang, paolo, jackie, fang, paolo, jackie, fang, paolo] ;
Path = [fang, paolo, jackie, fang, paolo, jackie, fang, paolo, jackie|...] ;
```

- Does it ever stop?

Some Queries

- Does it ever stop?
- Try to append `, false.` to the query:

```
?- ballfrom_reaches_lvia(jackie, paolo,Path), false.
% Hit Ctrl+C
Action (h for help) ? abort
% Execution Aborted
```

Some Queries

- Does it ever stop?
- The `,false` enforces backtracking, visiting all possible answers
- Finite solution space: interpreter returns `false`
  Remember: $P \land \bot \rightarrow \bot$ for any P in FOL
- Infinite answer sequence: infinite derivation (non-termination)
- Infinite set of answers leads to non-termination

Non-termination spreads

- Consider

```
ballfrom_reaches2(From, To) :-
    ballfrom_reaches_lvia(From, To, _Path).
```

  - Even worse:

```
?- ballfrom_reaches2(jackie, anne).
% unreachable, but explores infinitely long paths (non-termination)
```

Non-termination spreads

- Consider

```
ballfrom_reaches2(From, To) :-
    ballfrom_reaches_lvia(From, To, _Path).
```

- Execution:

```
?- ballfrom_reaches2(jackie, paolo).
true ;
true ;
true ;
true    % abort

?- ballfrom_reaches2(jackie, paolo), false.
% non-termination
```

  - Even worse:

```
?- ballfrom_reaches2(jackie, anne).
% unreachable, but explores infinitely long paths (non-termination)
```

Reordering goals can change termination behaviour

```
?- Xs = [something], isa_list(Xs).
Xs = [something].

?- isa_list(Xs), Xs = [something].
Xs = [something] ;
% loops enumerating all lists
```

Consequences of non-termination

- A recursive goal without instantiated variables always loops
- Putting always terminating predicates as early goals often helps (Restricts what gets passed to recursive goals)
- Properties of pure, monotonic Prolog programs (no non-logical elements, no negation)
  - Generalizing a non-terminating rule / query can not lead to termination e.g.: `isa_list([1,2,3|Xs])` to `isa_list(Xs)`
  - Instatiating a rule query can improve termination e.g. `isa_list([1,2,3|Xs])` to `isa_list([1,2,3|notalist])`
  - Removing goals from a rule can only increase the number of solutions

Outline

1. How to write a program
2. Execution of Prolog programs
3. Beyond Datalog: Lists
4. Two simple predicates over lists
5. Termination
6. Accumulators

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

```
aballfrom_reaches_lvia_acc(From,To,[To],Acc) :-
    child_throwsto(From,To),
    % ...
```

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

```
aballfrom_reaches_lvia_acc(From,To,[To],Acc) :-
    child_throwsto(From,To),
    % ...
aballfrom_reaches_lvia_acc(From,To,[Neighbour | Others], Acc) :-
    child_throwsto(From,Neighbour),
    % ...
    aballfrom_reaches_lvia_acc(Neighbour,To,Others, [Neighbour|Acc]).
```

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

```
aballfrom_reaches_lvia_acc(From,To,[To],Acc) :-
    child_throwsto(From,To),
    nonmember_of(To, Acc).
aballfrom_reaches_lvia_acc(From,To,[Neighbour | Others], Acc) :-
    child_throwsto(From,Neighbour),
    nonmember_of(From, Acc),
    aballfrom_reaches_lvia_acc(Neighbour,To,Others, [Neighbour|Acc]).
```

Hiding the accumulator

- This is still too general:

```
?- aballfrom_reaches_lvia_acc(jackie, paolo, Path, Acc).
Path = [fang, paolo],
Acc = [] ;
Path = [fang, paolo],
Acc = [_1204],
dif(_1204, paolo),
dif(_1204, jackie) ;
% ...
```

Hiding the accumulator

- This is still too general:

```
?- aballfrom_reaches_lvia_acc(jackie, paolo, Path, Acc).
Path = [fang, paolo],
Acc = [] ;
Path = [fang, paolo],
Acc = [_1204],
dif(_1204, paolo),
dif(_1204, jackie) ;
% ...
```

- Accumulator can have an arbitrary tail – start with an empty Acc

Hiding the accumulator

- Accumulator can have an arbitrary tail – start with an empty Acc

```
?- aballfrom_reaches_lvia_acc(jackie, paolo, Path, []).
Path = [fang, paolo] ;
false.

?- aballfrom_reaches_lvia_acc(jackie, anne, Path, []).
false.
```

Hiding the accumulator

- Accumulator can have an arbitrary tail – start with an empty Acc

```
?- aballfrom_reaches_lvia_acc(jackie, paolo, Path, []).
Path = [fang, paolo] ;
false.

?- aballfrom_reaches_lvia_acc(jackie, anne, Path, []).
false.
```

- Hide the accumulator from the user

```
aballfrom_reaches_lvia(From, To, Neighbour) :-
    aballfrom_reaches_lvia_acc(From, To, Neighbour, []).
```

Hiding the accumulator

- The new predicate always terminates:

```
?- aballfrom_reaches_lvia(From, To, Path), false.
false.
```

Summary

- Prolog's execution order weakens logical properties
- Swapping the order of goals does not influence the set of solutions but termination properties may change
- Appending `false` is a simple check for termination
- Narrowing answer set is an easy way to improve termination properties
- Accumulators pass information about the current goals to recursive goals

That's all for today!