AI Programming in PROLOG

2. Programming Patterns and code optimization
Recursion

• Tail Recursion:
  Schema: \( f(x) = \text{if } c(x) \text{ then } g(x) \text{ else } f(h(x)) \)
  PROLOG implementation:
  \[
  \begin{align*}
  f(X, Z) & :- c(X), g(X, Z).
  f(X, Z) & :- h(X, Y), f(Y, Z).
  \end{align*}
  \]

• Non-tail Recursion:
  – the value of the recursive call is modified after its computation.
  Schema: \( f(x) = \text{if } c(x) \text{ then } g(x) \text{ else } k(x, f(h(x))) \)
  PROLOG implementation:
  \[
  \begin{align*}
  f(X, Y) & :- c(X), g(X, Y).
  f(X, Y) & :- h(X, X1), f(X1, Y1), k(X, Y1, Y).
  \end{align*}
  \]
Space complexity of recursion

• Non-tail recursion:
  - Example:
    \[ f(n) = \text{if } n = 0 \text{ then } a \text{ else } k(n, f(n - 1)) \]
  - PROLOG implementation:
    \begin{align*}
    f(0, a). \\
    f(X, Y) & : - X1 \text{ is } X - 1, f(X1, Y1), k(X, Y1, Y).
    \end{align*}
  - Stack frames for \( f(N,a) \):
    \begin{align*}
    f(N,Y), & f(N-1,Y1), f(N-1,Y1_1), \ldots, f(1,Y1_N), f(0,a) \\
    f(N,Y), & f(N-1,Y1), f(N-1,Y1_1), \ldots, k(0,a,Y1_N) \\
    \vdots \\
    f(N,Y), & f(N-1,Y1), k(N-1,a,Y1) \\
    f(N,Y), & k(N,a,Y)
    \end{align*}
  - \( O(N) \) Space complexity

• Tail recursion:
  - If the recursive call is the last predicate in the bodies, then Space is \( O(1) \) with Last call optimization.
  - Transformation from Non-tail to Tail recursion is sometimes possible
Non-tail recursion transformation

• Non-tail recursion
  Schema: \( f(n) = \text{if } n = 0 \text{ then } a \text{ else } k(n, f(n - 1)) \)
  \[ f(0, a). \]
  \[ f(N, Y) :- N1 \text{ is } N - 1, f(N1, Y1), k(N, Y1, Y). \]

• Tail-recursion with accumulator
  \[ f(N, Y) :- f(N, 1, a, Y). \]
  \[ f(N, M, ACC, ACC) :- M > N. \]
  \[ f(N, M, ACC, Y) :- \]
  \[ M \leq N, \]
  \[ k(M, ACC, ACC1), \]
  \[ M1 \text{ is } M + 1, \]
  \[ f(N, M1, ACC1, Y). \]
Negation in Logic and PROLOG

• In Logic:
  - Positive knowledge and negative knowledge is explicitly asserted.
  - A theory states what is true and what is false.
  - If something cannot be proved that is true, it must (in principle) be considered neither true or false...
    • Unless the theory has the axiom of “tertium non datur”:
      - Forall A: A & Not(A) => false

• In PROLOG:
  - Only positive knowledge is explicitly asserted
  - Negative knowledge can be only “inferred”...
    • By means of “absence” of positive knowledge:
    • Something is false if it is not possible to prove that it is true.
Negation as Failure

• Negation (as failure or by default):
  – Try & Fail procedural model:
    • The goal succeeds if the negated predicate fails
  – NAF is not as Logical negation
    • Logical negation is monotonic
      – If P in Theory is true P is also true in Theory+{B} is also true.
  – NAF is “non-monotonic”
    • Example:

      \[
      \begin{array}{c}
      a :- b, \neg c. \\
      b. \\
      \neg a \\
      \text{yes}
      \end{array}
      \begin{array}{c}
      a :- b, \neg c. \\
      b. \\
      c. \\
      \neg a \\
      \text{no}
      \end{array}
      \]
Non-monotonic inference (dealing with “exceptions”)

• A PROLOG program about birds:

```prolog
flies(X) :- bird(X), \+ abnormal(X).
bird(tom).
bird(sam).
bird(donald).
abnormal(donald).
abnormal(X) :- isa(X; penguin).
isa(sam; eagle).
isa(tom; penguin).
isa(donald; duck).
```

• Running (querying) the program:

```prolog
?- bird(X).
X = tom ;
X = sam ;
X = donald ;
No
?- flies(X).
X = sam ;
No
```
Program Optimization

- PROLOG is essentially a database query engine!
  - More powerful than SQL because it is Turing-complete
    - => By default, it tries to find all the answers for a query.
  - Sometimes, one needs to know only if the answer is positive, but avoid searching for all the instances:
    - Existential queries:
      - Example: a man is a father if he has AT LEAST one kid.
    - Due to backtracking, PROLOG “blindly” executes queries that have already been executed.

- Default computational mechanism in PROLOG:
  - Generate & Test:
    - => Exponential Time!!
  - Trick:
    - PRUNE the Search Tree to avoid looking at redundant solutions
Cut

• PROLOG offers mechanisms for program optimization:
  - One we have already seen:
    • Rules and predicate orders in program.
  - One that affect the program execution:
    • CUT

• The cut is a predicate “!” that always succeeds:
  ⇒ it does not alter the logical semantics of the program.
  • Introduces a “side-effect“:
    • The goals in the clause before the cut are no longer executed;
    • including, importantly, the choice of using that particular clause!
Problems with cut (1)

• Completeness:
  - Solutions exists, but the program does not find them
  - Example:
    
    proud(X) :- father(X,Y), newborn(Y).
    father(X,Y) :- parent(X,Y), male(X),!.
    parent(john, mary).
    parent(john, chris).
    male(john).
    newborn(chris).
Red and Green cuts

- Two principal uses of cut:
  - **GREEN**: cut failing or redundant branches from SLD tree:
    - Preserves the program semantics
    - Improves performance
  - **RED**: prune non-redundant succeeding branches of SLD tree:
    - Leads to incompleteness or unsoundness.
    - Difficult to detect!

- "Cut" makes of PROLOG an efficient programming language...
  - at the expenses of pure logical semantics
Examples of green cuts: list merging

merge([X|Xs], [Y|Ys], [X|Zs]) :- X < Y, !, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,Y|Zs]) :- X = Y, !, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X > Y, !, merge([X|Xs], Ys, Zs).
merge(Xs, [], Xs) :- !.
merge([], Ys, Ys) :- !.

merge([1,3,5], [2,3], Zs).

1 < 2, !, merge([3,5], [2,3], Zs0).
   Zs = [1|Zs0]
1 = 2, !, merge([3,5], [3], Zs0).
   Zs = [1,2|Zs]
1 < 2, !, merge([3,5], [2,3], Zs0).
   Zs = [2|Zs]

merge([3,5], [2,3], Zs).

3 < 2, !, merge([5], [2,3], Zs1).
   Zs0 = [3|Zs1]
3 = 2, !, merge([5], [3], Zs1).
   Zs0 = [3,2|Zs1]
3 > 2, !, merge([3,5], [3], Zs1).
   Zs0 = [2|Zs1]

merge([3,5], [3], Zs1).

3 < 3, !, merge([5], [3], Zs2).
   Zs1 = [3|Zs2]
3 = 3, !, merge([5], [], Zs2).
   Zs1 = [3,3|Zs2]
3 > 3, !, merge([3,5], [], Zs2).
   Zs1 = [3|Zs2]

merge([5], [], Zs2).
   Zs2 = [5]
Green cuts for single-solution predicates

- **Membership:**
  - Avoid computing membership for duplicates.
  - Whenever the element is found, then stop.
  - Naive Program:
    
    ```
    member(X, [X | L]).
    member(X, [Y | L]) :- member(X, L).
    
    With cut:
    member(X, [X | L]) :- !.
    ```

- **Adding elements to Lists with no-duplicates:**
  - Insert an element only if does not already belong to the list
  - Program:
    
    ```
    add(X, L, L) :- member(X, L), !.
    add(X, L, [X|L]).
    ```
Example of red cut

• The minimum predicate with intended logical semantics:
  \[
  \text{min}(X,Y,X) :- X \leq Y. \\
  \text{min}(X,Y,Y) :- X \geq Y.
  \]

• With “naive” cut:
  \[
  \text{min}(X,Y,X) :- X \leq Y, !. \\
  \text{min}(X,Y,Y). \\
  \text{– it works on min}(2,5,X), but not on min(2,5,5)!!}
  \]

• Safe version:
  \[
  \text{min}(X,Y,Z) :- X \leq Y, !, Z=X. \\
  \text{min}(X,Y,Y).
  \]
Negation as Cut&Fail

• "fail" is a predicate that always fails.
• It can be used in combination with cut to implement Negation as Failure:
  
  not(Goal) :- call(Goal),!,fail.
  not(Goal).
  
  - "not/1" is a predicate that takes as parameter a term.
  - "call" is a meta-predicate that executes a term as it was a predicate.

• It is better to avoid Cut&Fail negation and rely on the built-in negation predicate "\+".
Weird interactions between cut and negation

• Simple programs:
  
  p :- a, b.
  p :- c.
  Logical Semantics: p ⇔ (a ∧ b) ∨ c

  p :- a,!b.
  p :- c.
  Logical Semantics: p ⇔ (a ∧ b) ∨ (¬a ∧ c)

  p :- c.
  p :- a,!b.
  Logical Semantics: p ⇔ c ∨ (a ∧ b)

• Inverting the order of clauses with cuts affects the logical semantics of the program!!!
  
  => Be very careful in using cuts!
PROLOG and CWA

• PROLOG adopts the Closed World Assumption:
  - Something “exists” if and only if it stated in the program.
  - This is a consequence of NAF.

• Truth vs Knowledge
  - In PROLOG they are essentially the same
  - Something is true if and only if it is known (declared).

• Reality is different!!!
  - Open World Assumption
  - If we are not able to conclude that something is true, it does not
    mean that it must be false.
  - It means that we are just not smart enough or we lack
    knowledge...
PROLOG and Predicate Logic

- Quantifiers are difficult to model in PROLOG
  - Essentially because of NAF

- In logic:
  - For all A in G: C ↔ not Exists A in G: not(G)

- In PROLOG:
  - `forall(G,C) :- not(G, not C).`
  - Works with some restrictions...
  - Which ones?
  - G,C are meta-variables!
    - “forall” is a meta-predicate
  - Can we define Exists?
    - `exists(G,C) :- not(forall(G, not(C))).`
Difference Lists

• Lists are recursive structures:
  – => Operations typically need to traverse the structure
  • E.g. append:
    
    ```prolog
    append([], L, L).
    append([X | L1], L2, [X | L3]) :- append(L1, L2, L3).
    ```

• Trick: smart use of built-in unification!!
  – Open lists: \([a, b, c | X]\)
    • Represents lists with whatever tail
    
    ```prolog
    append([a, b, c | X], [d, e, f | Y], [a, b, c, d, e, f | Z])
    ```
    – A predicate that unifies \([d, e, f]\) with \(X\) and \(Y\) with \(Z\) would make the work!

  – Difference lists: \([a, b, c | X] - X\)
    • “Represents” the instantiated part of the list, but keeping the open list representation.
Operations with difference lists

• Append:

\[
\text{diff_app}(X-Xh, Y-Yh, Z-Zh) :- Z=X, Y=Xh, Yh=Zh.
\]
\[
\text{diff_app}([a,b,c|A]-A, [e,f|B]-B, C) =>
Z=[a,b,c|A], Y=[e,f|B]=A, Yh=B=Zh,
C=Z-Zh=[a,b,c,e,f|B]-B.
\]

Simplified version:

\[
\text{diff_app}(X-H1, H1-H2, X-H2).
\]
Operations with difference lists

- **Insert:**
  
  \[
  \text{ins\_first}(X-Y, E, Z) :\quad \text{diff\_app}([E|B]-B, X-Y, Z).
  \]
  \[
  \text{ins\_first}(X-Y, E, [E|H]-Y) :\quad X = H.
  \]
  \[
  \text{ins\_first}(H-Y, E, [E|H]-Y).
  \]
  
  \[
  \text{ins\_last}(X-Y, E, Z) :\quad \text{diff\_app}(X-Y, [E|B]-B, Z).
  \]
  \[
  \text{ins\_last}(X-Y, E, X-H) :\quad Y = [E|H].
  \]
  \[
  \text{ins\_last}(X-[E|H], E, X-H).
  \]

- **Delete:**
  
  - Any ideas?
  - Is it possible?
  - ...try!!