Programming in Logic: Prolog

Negation
Mode Specifications
Gen&Test Transformations
Hierarchical data structures and data abstraction are important for making Prolog programs easy to understand and to modify.

When one doesn’t know how to generate a solution to a problem directly, it’s often useful to view the problem as generating candidate solutions and then testing that they satisfy all the success criteria.
Review cont’d

- The generation phase finds values for variables in the solution structure.
- The test phase checks that values are “correct”.
- The phases can often be broken into parts.
- These parts are then implemented as simply as possible.
- These programs are naive & often need to be incrementally transformed into efficient ones.
Negation: Not/1

- As we mentioned in earlier lectures, Prolog does not handle true negation.

- Pure Prolog does not even have an explicit way of expressing negation.

- The negation that Prolog does have is a negation by failure-to-prove.
not/1

not(P) : $P$ cannot be a variable, it must be fully instantiated to guarantee it behaves as expected.

We will not discuss how not/1 is implemented at this point, we will cover that later.
Using \textit{not/l}

- Defns of relations that use $\text{not}(P)$ must guarantee that $P$ is instantiated.

- If $P$ has come from user’s query, it may not be possible to guarantee it has been instantiated.

- Relation’s documentation must specify that the argument needs to be instantiated.
Argument Modes

- There is a standard way of specifying whether a relation’s arguments must be instantiated, uninstantiated, or doesn’t matter.
- This is done by specifying the argument’s mode.
- In the relation’s documentation, the argument’s modes are usually specified:
  - `relationname(arg1ModeSpec, ...)`
Mode Specification

- The argument mode specification is simply the name/role of argument preceded by its mode.
- The modes are:
  - +: must be instantiated
  - -: must be uninstantiated
  - ?: can be either
- Examples:
  - not(+Goal)  member(?Elt, ?List)
Quick Check

Given relation below, what’s wrong with it?

/* bachelor(?Person) */
bachelor(P):-
    not(married(P)),
    male(P).

How can we fix it?
Last Homework Assignment

Lisa looked in the newspaper to see what movies were playing at local cinema. She was delighted to see that four films starring her four favorite actors are now showing. Each film is a different type - drama, horror, comedy, and science fiction. From this information, and the following clues, your program should find the title of each type of film Lisa wants to see (one is titled *Choice*), and the star of each film (one is Doug Drew).
The Clues

The clues are: The comedy does not star Bob Apples, who stars in *Shoe*. *Boom!* is a science fiction tale. Lisa finally decided to go see the drama starring Anna Graham. Carl Cool is not the star of *Doggone*, which is the comedy.

Try writing a Prolog program that solves this puzzle. You can use the *distinct* relation.
Naïve Gen&Test Solution
Solution Structure

- First step is to decide on solution structure.
- The solution structure must tell the title, star, and genre for the four movies.
- One solution structure is list of “movie info” items, where each movie info item is a structure containing movie’s title, star, and genre.
movies(Movies) :-
\% GENERATE CANDIDATES
  titles(Titles),
  stars(Stars),
  genres(Genres),
  Movies = [m(M1T,M1S,M1G),
            m(M2T,M2S,M2G),
            m(M3T,M3S,M3G),
            m(M4T,M4S,M4G)],
  distinctElts(M1T, M2T, M3T, M4T, Titles),
  distinctElts(M1S, M2S, M3S, M4S, Stars),
  distinctElts(M1G, M2G, M3G, M4G, Genres), ...

Movie Info Domains
  titles([choice, shoe, boom, doggone]).
  stars([dougDrew, bobApples, annaGraham, carlCool]).
  genres([drama, horror, comedy, scifi]).
distinctElts/5

distinctElts(E1, E2, E3, E4, List) :-
    member(E1, List),
    member(E2, List),
    member(E3, List),
    member(E4, List),
    distinct(E1, E2, E3, E4).

/* distinct(+Item1, +Item2, +Item3, +Item4) */
distinct(A,B,C,D) :-
    not(A=B), not(A=C), not(A=D), not(B=C), not(B=D), not(C=D).
Naïve Gen&Test Solution
Candidate Tests

Movies(Movies) :- . . .
% TEST CANDIDATES
member(m(ComedyTitle, ComedyStar, comedy), Movies),
not(ComedyStar = bobApples),
member(m(shoe,bobApples,ShoeGenre), Movies),
not(ShoeGenre = comedy),
member(m(boom, _, scifi), Movies),
member(m(_, annaGraham, drama), Movies),
not(ComedyStar = carlCool),
ComedyTitle = doggone.

Clues
The comedy does not star Bob Apples, who stars in Shoe. Boom! is a science fiction tale. Lisa finally decided to go see the drama starring Anna Graham. Carl Cool is not the star of Doggone, which is the comedy.
Incremental Program Modifications

- Only be looking at a few simple ones.
- Won’t consider ordering of clauses.
- Will focus on ordering/transformations of goals.
- Look at 3 types of transformations:
  - Factoring
  - Merging
    - Gen-Gen, Test-Test, Test-Gen
  - Specialisation
Gen&Test Transformations
Test Movement

- If a binding is wrong, then all computation between its generation and its test is wasted.

- Want to test binding generation as early as possible.

- In pure Prolog, do this by moving test goals earlier in clause.
### Test Movement Example

<table>
<thead>
<tr>
<th>Left Side</th>
<th>Right Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{member}(m(\text{CTitle}, \text{CStar}, \text{comedy}), \text{Movies})</td>
<td>\text{member}(m(\text{CTitle}, \text{CStar}, \text{comedy}), \text{Movies})</td>
</tr>
<tr>
<td>\text{not}(\text{CStar} = \text{bobApples}), \text{member}(m(\text{shoe}, \text{bobApples}, \text{ShoeG}), \text{Movies}), \text{not}(\text{ShoeG} = \text{comedy}), \text{member}(m(\text{boom}, _ , \text{scifi}), \text{Movies}), \text{member}(m(_ , \text{annaGraham}, \text{drama}), \text{Movies}), \text{not}(\text{CStar} = \text{carlCool}), \text{CTitle} = \text{doggone}.</td>
<td>\text{member}(m(\text{shoe}, \text{bobApples}, \text{ShoeG}), \text{Movies}), \text{not}(\text{ShoeG} = \text{comedy}), \text{member}(m(\text{boom}, _ , \text{scifi}), \text{Movies}), \text{member}(m(_ , \text{annaGraham}, \text{drama}), \text{Movies}).</td>
</tr>
</tbody>
</table>
Ordering Tests

- Tests should be ordered based on how much they will reject.

- The more “specialized” the test, the earlier it should appear.
Test Ordering Example

\[
\text{member}(m(CTitle, CStar, comedy), Movies), \\
\text{not}(CStar = \text{bobApples}), \\
\text{not}(CStar = \text{carlCool}), \\
\text{CTitle} = \text{doggone}, \\
\text{member}(m(\text{shoe, bobApples, ShoeG}), Movies), \\
\text{not}(\text{ShoeG} = \text{comedy}), \\
\text{member}(m(\text{boom, }, \text{scifi}, Movies), \\
\text{member}(m(_, \text{annaGraham, drama}), Movies).}
\]
Test Merging

- Sometimes separate tests can be merged into a single test goal.

```
member(m(CTitle, CStar, comedy), Movies),
CTitle = doggone,
```

```
member(m(doggone, CStar, comedy), Movies),
```
Test Merging Example

\[
\text{member}(m(\text{CTitle}, \text{CStar}, \text{comedy}), \text{Movies}),
\]
\[
\text{CTitle} = \text{doggone},
\]
\[
\text{not}(\text{CStar} = \text{bobApples}),
\]
\[
\text{not}(\text{CStar} = \text{carlCool}),
\]
\[
\text{member}(m(\text{shoe}, \text{bobApples}, \text{ShoeG}), \text{Movies}),
\]
\[
\text{not}(\text{ShoeG} = \text{comedy}),
\]
\[
\text{member}(m(\text{boom}, _, \text{scifi}), \text{Movies}),
\]
\[
\text{member}(m(_, \text{annaGraham}, \text{drama}), \text{Movies}).
\]

\[
\text{member}(m(\text{doggone}, \text{CStar}, \text{comedy}), \text{Movies}),
\]
\[
\text{not}(\text{CStar} = \text{bobApples}),
\]
\[
\text{not}(\text{CStar} = \text{carlCool}),
\]
\[
\text{member}(m(\text{shoe}, \text{bobApples}, \text{ShoeG}), \text{Movies}),
\]
\[
\text{not}(\text{ShoeG} = \text{comedy}),
\]
\[
\text{member}(m(\text{boom}, _, \text{scifi}), \text{Movies}),
\]
\[
\text{member}(m(_, \text{annaGraham}, \text{drama}), \text{Movies}).
\]
Removing Redundant Tests

- Sometimes there are redundant tests, these can be eliminated.
- The generator already ensures that distinct choices have been made for each movie.
- Tests which duplicate this are redundant and can be removed.
Removing Redundant Tests

Example

\[
\text{member}(m(\text{doggone}, CStar, \text{comedy}), \text{Movies}), \\
\quad \text{not}(CStar = \text{bobApples}), \\
\quad \text{not}(CStar = \text{carlCool}), \\
\quad \text{member}(m(\text{shoe}, \text{bobApples}, \text{ShoeG}), \text{Movies}), \\
\quad \text{not}(\text{ShoeG} = \text{comedy}), \\
\quad \text{member}(m(\text{boom}, _, \text{scifi}), \text{Movies}), \\
\quad \text{member}(m(_, \text{annaGraham}, \text{drama}), \text{Movies}).
\]
movies(Movies) :-
    titles(Titles), stars(Stars), genres(Genres),
    Movies = [m(M1T,M1S,M1G), m(M2T,M2S,M2G),
    m(M3T,M3S,M3G), m(M4T,M4S,M4G)],
    distinctElts(M1T, M2T, M3T, M4T, Titles),
    distinctElts(M1S, M2S, M3S, M4S, Stars),
    distinctElts(M1G, M2G, M3G, M4G, Genres),
    member(m(doggone, CStar, comedy), Movies),
    not(CStar = carlCool),
    member(m(shoe,bobApples,_), Movies),
    member(m(boom, _, scifi), Movies),
    member(m(_, annaGraham,drama),Movies).
Test Incorporation

- Sometimes, generator and a test are formulated such that the test can be folded into generator.
- For example, assume that $x$ and $y$ are positive integers such that $x > 4$ and $x + y = 7$.
- Naïve version (pseudo-Prolog):
  - `member(X, I), member(Y,I), X>4, X + Y = 7`
- Test Incorporated Version (pseudo-Prolog):
  - `member(X,I), X>4, Y = 7 - X`
movies(Movies) :-
titles(Titles), stars(Stars), genres(Genres),
Movies = [m(M1T,M1S,M1G), m(M2T,M2S,M2G),
m(M3T,M3S,M3G), m(M4T,M4S,M4G)],
member(m(doggone, CStar, comedy), Movies),
member(CStar, Stars),
not(CStar = carlCool),
member(m(shoe,bobApples,ShoeG), Movies),
member(ShoeG, Genres),
member(m(boom, SStar, scifi), Movies),
member(SStar, Stars),
member(m(DTitle, annaGraham,drama),Movies),
member(DTitle, Titles),
distinct(M1T, M2T, M3T, M4T),
distinct(M1S, M2S, M3S, M4S),
Too General A Generator

- Sometimes, a relation defn leaves certain choices unconstrained.
- These extra degrees of freedom mean that a lot more things match our definition.
- Our example has 24 different answers, which only differ in order items appear in list.
- This generally means that a lot of work goes to show that something does not satisfy the defn.
Specializing a Generator

- When all choices are equally acceptable, one can be chosen arbitrarily to be “right” answer.

- In our example, we could chose that the movies would in genre alphabetical order in the list.

- This specializes the relation being defined.

- Some movie info lists that satisfied the old relation defn will no longer do so.

- This reduces “cost” of computing the relation.
moviesInGenreOrder(Movies) :-
titles(Titles), stars(Stars),
Movies = [m(doggone, CStar, comedy),
    m(DTitle, annaGraham, drama),
    m(HTitle, HStar, horror),
    m(boom, SStar, scifi)],
member(CStar, Stars),
not(CStar = carlCool),
member(SStar, Stars),
member(HStar, Stars),
distinct(CStar, annaGraham, HStar, SStar),
member(m(shoe, bobApples, _), Movies),
member(DTitle, Titles),
member(HTitle, Titles),
distinct(doggone, DTitle, HTitle, boom).

**Warning:** Name of relation has changed to show that this relation is not the same as the old one.
What’s Changed?

- Is this a better program?
  - How has the clarity changed?
  - How has the efficiency changed?
  - How has the generality changed?
Summary

- Use the `not/1` relation to express negation, but its argument has to be instantiated to behave properly.

- `not/1` expresses failure-to-prove, not actual falsity, but usually close enough if above constraint satisfied.
Summary cont’d

- Documentation of relation should show the modes for its arguments.
- Mode describes whether the argument should be instantiated, uninstantiated, or doesn’t matter when the relation is “called”.
- Use mode declarations to work out whether variable must be instantiated when goal called.
Summary cont’d

There are various transformations of generators / testers that increase efficiency:

- Test movement
  - Close to generator
  - Most constraining test first
- Test merge
- Test incorporation (into generator)
- Generator specialization