Programming in Logic: Prolog

Negation cont’d
Knights & Knaves
Negation & Variables

• Signature for \( \text{not} \) is \( \text{not}(+P) \).

• Why does \( \text{not}/1 \) require its argument to be instantiated?

• What happens if it has unbound variables in its argument?
Negation & Unbound Variables

**Example:**
Assume we define $bachelor/1$ to be:

$bachelor(X) :- not(married(X)), male(X)$.

Assume the KB contains the following facts:

married(john).

male(john).

male(kurt).

What will Prolog do with the query: $bachelor(B)$

First, it will try goal $not(married(B))$, what happens?
The goal \texttt{not(married(B))} fails.

Definition of \texttt{not/1}:
\[
\text{not}(P) :- P, !, \text{fail}.
\text{not}(_).
\]

Call

\texttt{married(B)}

\texttt{married(john)}

\texttt{true}

\texttt{!}

\texttt{fail}

Fail

\texttt{true}
What happened to $bachelor(B)$?

- $bachelor(X) :- not(married(X))$, $male(X)$.

- The goal $not(married(B))$ fails therefore the query $bachelor(B)$ fails.
Variables and Negation

• \( \text{not}(P) :- P,!,\text{fail}. \)
  
  \( \text{not}(\_). \)

• Note: variables in \( P \) only get bound by \( P \) matching something in KB.

• But if a match occurs then \text{not/1} fail.

• Therefore no variables in \( P \) can get bound by calling \( \text{not}(P) \)
Variables and Negation cont’d

• When does not\((P)\) ever succeed?
• Only when there is no way to prove \(P\)!
• Problem is we look at defn of bachelor and read it declaratively, i.e., is there an \(X\) such that \(X\) is both unmarried and a male.
• What do we do to our defn of bachelor to enable it to compute bachelors?
bachelor/1 Revisited

• bachelor(X) :- male(X), not(married(X)).
• Now, X will eventually bind to kurt who Prolog will “recognize” as a bachelor.
• male/1 is our generator that makes sure that X is bound (if it can be) before calling the test not/1 tester.
Knights & Knaves

Every inhabitant is either a knight or a knave. Knights always tell the truth and knaves always lie. Three inhabitants were standing together in a garden. A stranger passed by and asked A, “Are you a knight or a knave?” A answered, but rather indistinctly, so the stranger could not make out what he said. The stranger then asked B, “What did A say? B replied, “A said that he is a knave.” At this point the third man, C, said, “Don’t believe B; he is lying!” What are B and C?
Attacking the Problem

• How does one approach trying to write a Prolog program that solves this puzzle?
• The first thing to note is that we probably don’t know of a direct way to solve this problem, instead, we’re going to have to search for a solution.
• One approach is our old friend “generate&test”.
Steps in Gen & Test

• Gen & test assumes that can enumerate all possible solution candidates & test each one to see which are actual solutions.
• Program develops through a sequence of steps, come up with:
  – A solution structure
  – How to generate all possible candidates
  – How to determine if candidate is actual solution
Solution Structure

• What question are you asked to answer?
• Are B and C knights or knaves?

• Extend the structure to A, B, and C.
A K&K Solution Structure

• List of identifications (knight or knave):
  – [A_Id, B_Id, C_Id]

• What’s the structure of an identification?
  – Says what someone is:
    \[ \text{isa}(	ext{Person}, \text{Type}) \] where \text{Person} is someone in puzzle and \text{Type} is knight or knave.
  – Example: \( \text{isa}(b, \text{knight}) = b \) is a knight.
Solution Structure

• Now we have our solution structure:
  \[ \text{isa}(a, A), \text{isa}(b, B), \text{isa}(c, C) \]
  where the A, B, and C will be eventually bound to either \textit{knight} or \textit{knave}.

• Now we can write a generator for our solution structure.
Solution Generator

```prolog
solve(Clues, Solution) :-
    Solution = [isa(a,A), isa(b,B), isa(c,C)],
    Types = [knight, knave],
    member(A, Types),
    member(B, Types),
    member(C, Types) ...
```
Next Step

• What do we do next?

• Our next step is to try to interpret the clues so that they test our generator’s knight/knave assignments.
Normalizing The Clues

• Want the clues in a similar format.

• What are the clues:
  – B replied, “A said that he is a knave.”
  – C said “Don’t believe B; he is lying!”

• Interpret the clues to be about knights/knaves.

• The first clue is straightforward
  – B says A says A is a knave.

• The second clue can be understood as:
  – C says that B is a knave.
Encoding the Clues

- Clues have form: “someone says something”:
  (a) someone saying someone says ...;
  or (b) someone is a knight/knave.

- Can encode someone saying something as:
  \[
  \text{says(Person, Statement)}
  \]

- Clues become:
  - \[
    \text{says(b, says(a, isa(a, knave)))}
  \]
  - \[
    \text{says(c, isa(b, knave))}
  \]
Interpreting the Clues

• Need to interpret the clues as constraints on the solution structure.

• Clues: statements about who says who is what.

• Can view clues as providing consequences:
  – If knight says X is a Y then X is a Y

• Candidate solution must be consistent with clue’s consequences:
  – consequence = X is Y ⇒ candidate must have X is Y
solve(Clues, Solution) :-
    Solution = [isa(a,A), isa(b,B), isa(c,C)],
    Types = [knight, knave],
    member(A, Types),
    member(B, Types),
    member(C, Types),
    consequences(Solution, Statements, Consequences),
    consistent(Solution, Consequences).
Interpreting the Clues cont’d

• In order to know who’s a knight/knave, need to look at current candidate solution.

• $\text{consequences(}+\text{Solution, } +\text{Clues, } -\text{Consequences)}$

• Examples of consequence derivation:
  – Knight says $X$ is knight $\Rightarrow X$ is knight
  – Knave says $X$ is knight $\Rightarrow X$ is knave
  – Knight says $X$ says $Y$ is $Z$ $\Rightarrow X$ says $Y$ is $Z$
  – Knave says $X$ says $Y$ is $Z$ $\Rightarrow \text{no consequence}$
Encoding Consequences

Examples:

\[
\text{consequences}(\text{Solution}, [\text{says}(X, \text{isa}(Y, \text{knave}))|Ss], [C|Cs]) : - \\
\text{member}(\text{isa}(X, \text{knave}), \text{Solution}), \\
C = \text{isa}(Y, \text{knight}), \\
\text{consequences}(\text{Solution}, Ss, Cs).
\]

\[
\text{consequences}(\text{Solution}, [\text{says}(X, \text{says}(Y, \text{Statement}))|Ss], \text{Consequences}) : - \\
\text{member}(\text{isa}(X, \text{knight}), \text{Solution}), \\
\text{consequences}(\text{Solution}, [\text{says}(Y, \text{Statement})|Ss], \text{Consequences}).
\]

\[
\text{consequences}(\text{Solution}, [\text{says}(X, \text{says}(_, _))|Ss], \text{Consequences}) : - \\
\text{member}(\text{isa}(X, \text{knave}), \text{Solution}), \\
\text{consequences}(\text{Solution}, Ss, \text{Consequences}).
\]
Interpreting Consistency

• What does it mean for the candidate solution to be consistent with the consequences?

• It means that nothing in the consequences can contradict the candidate solution, for example:
  – If $a$ is knight in candidate then one of consequences cannot be that $a$ is knave, and vice versa.
Encoding Consistency

consistent([], _).

consistent([isa(X,knight)|Sols], Consequences) :-
    not(member(isa(X,knave), Consequences)),
    consistent(Sols, Consequences).

consistent([ isa(X,knave)|Sols], Consequences) :-
    not(member(isa(X,knight), Consequences)),
    consistent(Sols, Consequences).