Applications of Artificial Intelligence

Sebastian Iwanowski
FH Wedel

Chapter 2:

Logic- and Rule-Based Programming
Using the Example of Prolog

Literature for Prolog

Textbooks:

Ivan Bratko: *PROLOG, Programming for Artificial Intelligence*,
2nd Edition, Pearson 1990, ISBN 0-201-41606-9
3rd Edition, Pearson 2001, ISBN 0-201-40375-6
4th Edition, Pearson 2011, ISBN 0-321-41746-6
Companion website with Prolog code: www.pearsoned.co.uk/bratko

P. Blackburn, J. Bos, K. Striegnitz: *Learn Prolog Now!*,
Texts in Computing Vol. 7, King's College Publications. 2006, ISBN 1-904987-17-6.
Companion website with on-line version: www.learnprolognow.org

Peter Bothner / Wolf-Michael Kähler: Programmieren in *PROLOG (in German),*
Eine umfassende praxisgerechte Einführung,
Vieweg 1991, ISBN 3-528-05158-2

Seminar presentation (in German):

Max Rohde: *Eignung logischer Programmiersprachen für Spiele-KI am Beispiel Prolog,*
FH Wedel, Iwanowski, SS 2007, Informatik-Seminar zur Spiele-KI
↳ gibt auch einen Überblick über Prolog und enthält weiterführende Literaturliste

Elements of PROLOG

Elementary components:

- **numbers**

  Integer and real numbers are distinguished ($1 \neq 1.0$).

- **atoms**

  name where the first character is a small literal

- **variables**

  name where the first character is a capital literal, exception: _

- **lists**

  [] or [term | list]
  short notation: [1,2,3,4] for [1 | [2 | [3 | [4 | []]]]]

- **terms**

  numbers, atoms, variables, lists or expressions like atom(term), atom(term,term) or ...

- **predicates**

  terms of the type atom(term), atom(term,term) or ...
  2 predicates are equal, if their name is the same atom and the number of parameters is the same.

Elements of PROLOG

Logic operators between predicates:

- **conjunction**

  a , b corresponds to: a ∧ b

- **implication**

  a :- b corresponds to: b → a

- **equivalence**

  a = b corresponds to: b ↔ a

- **antiequivalence (exor)**

  a \= b corresponds to: b ↮ a

- **version-specific operators for comfort**

  member, length, ...

Elements of PROLOG

Arithmetic operators

- **+, -, *, /, div, mod**

  Arithmetic expressions are always formed in infix notation.

Evaluation of arithmetic expressions

- **not automatically!**

- **when a variable is assigned an expression**

  `varname` **`is`** `arithmetic expression`
  Result of the arithmetic expression is assigned to the variable.

- **using special logic operators with evaluation capability**

  <, =<, > >=. =:=, =\= evaluate arithemtic expressions on either side.
  (in some implementations only on one side)

Elements of PROLOG

Knowledge in form of clauses

- **facts**

  `predicate.`
  Such predicates are assumed to be true in the knowledge base.

- **rules**

  `predicate :- conjunction of predicates.`
  The concluding predicate (on the left) is considered true
  if the proposition (on the right) has to be assumed true.
  For the same concluding predicate there may be different rules.

- **queries**

  `?- conjunction of predicates.`
  Prolog tries to derive the truth of a query from the known facts and rules.
  If this derivation is successful, the answer is `yes` and the values
  necessary to bind on a variable for the verification are output.
  Otherwise the answer is `no`.

Elements of PROLOG

Prolog's special handling of *not*

- **Most versions of Prolog provide a concept for negation**

  `not` Term
  `\+` Term
  Term1 `=\=` Term2

  Prolog evaluates these predicates to true if it cannot prove that Term is `true` resp. Term1 = Term2.

  Warning:

  This is not the same as that Prolog can prove that Term is `false` resp. Term1 $\neq$ Term2

  Consequence:

  Strict mathematical problem solvers better avoid using negation.

Functionality of a PROLOG interpreter

PROLOG is knowledge-based:

- **Knowledge base**

  Facts and rules, dynamically extensible

- **Inference engine**

  deriving facts and rules automatically using the inference techniques **resolution** und **unification**

- **Dialog component**

  Input: Query
  Output: yes / no, Specification of used unification in case of success, write as a „side effect“

  Yes: The predicate of the query can be concluded from knowledge base.
  No: The predicate of the query cannot be concluded from knowledge base.
  No does not imply that it can be concluded that the predicate is false.

Functionality of a PROLOG interpreter

How the inference engine works:

- **Decomposition of a goal into subgoals**

  First goal is the original query.
  Prolog tries to achieve the goal with unifications of the predicates of the knowledge base.
  This makes the predicates to subgoals.

- **Order of evaluation**

  All data of the knowledge base are evaluated **from top to bottom**.
  Conjunctions of rule propositions are evaluated **from left to right**.
  The evaluation order does *not* distinguish between facts and rules.

- **Instantiation of variables**

  Variables are instantiated with values only for the sake of unification.
  The current instantiation is removed after definite success or failure of unification with this value.

- **Backtracking**

  Failure of a unification automatically initiates a new instantiation.
  Deep backtracking: Try the verification with a different value in the proposition for the same rule.
  Shallow Backtracking: Try to verify a different rule implying the same predicate.

PROLOG: Simple example

- **Predicate world from first semester:**

Knowledge base:

```
father(sven,georg).
brother(holger,anna).
married(sven, anna).

male(X) :- father(X,Y).
male(X) :- brother(X,Y).

uncle(X,Y) :- father(Z,Y), brother(X,Z).
uncle(X,Y) :- mother(Z,Y), brother(X,Z).
mother(X,Y) :- father(Z,Y), married(X,Z).
female(X) :- married(X,Z), male(Z).
married(X,Y) :- married(Y,X).
```

Queries:

```
?-female(anna).
?-male(georg).
?-uncle(holger,georg).
?-male(X).
?-married(holger,X).
```

Declarative alternative without problems with symmetric predicates: XSB http://xsb.sourceforge.net/

In ISO-Prolog this does not work!

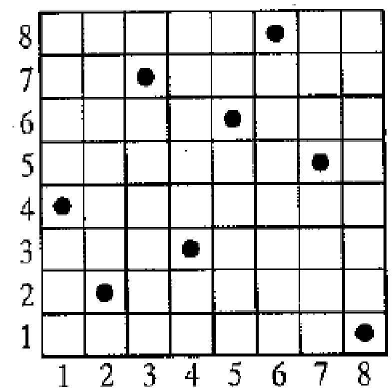
better:

```
isMarried(X,Y) :- married(X,Y).
isMarried(X,Y) :- married(Y,X).
```

```
?- isMarried(holger,X).
```

PROLOG: More complicated example

- **8 queens problem (1st solution of Bratko)**



Knowledge base:

```
queens1([]).

queens1([X/Y | Others]) :-
   queens1(Others),
   member(Y,[1,2,3,4,5,6,7,8]),
   conflictFree(X/Y,Others).

conflictFree(_,[]).

conflictFree(X/Y, [HeadX/HeadY | Others]) :-
   Y =\= HeadY,
   DiffY is HeadY - Y,
   DiffY =\= HeadX - X,
   DiffY =\= X - HeadX,
   conflictFree(X/Y,Others).

template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

not: DiffY =:= HeadY-Y
not: HeadY - Y =\= HeadX-X

Query:

query for a single answer:
?-template(S), queens1(S).

query for all answers:
?-template(S), queens1(S), write(S), nl, fail.

Base Technology: Logic Programming Language

- **Input:**
  Specification of the problem with a logical description language

- **Output:**
  Response in a logical description language

- **Automatically (without specifying algorithms!):**
  Generation of output from input

- **For improvement of efficiency:**
  Different specifications of the problem are possible and may influence the output if the automatic generation procedure is well-understood

Logic programming languages

Task for the interpreter:

less than ever not decidable for arbitrary formulae

Original goal: Construction task

Given a set $\mathscr{F}$ of logic formulae. Determine all formulae that can be logically derived from $\mathscr{F}$.

Easier goal: Verification task ***not decidable for arbitrary formulae***

Given a set $\mathscr{F}$ of logic formulae and a (new) logic formula F.
Find out if F can be derived from $\mathscr{F}$.

Problems equivalent to the verification task:

1) Given a set $\mathscr{F}$ of logic formulae and a (new) formula F. Find out if the set $\{\neg F\} \cup \mathscr{F}$ is contradictory.

2) Given a set $\mathscr{F}$ of logic formulae. Find out if it is contradictory.

Corresponds to satisfiability problem: not decidable for arbitrary formulae

Chances to simplify the problem:

Restrict the class of admissible formulae !

Propositional formulae

- **A propositional formula on truth values is a combination of finitely many literals with operators of propositional logics.**
  - The literals are variables which may assume exactly one of two values.

- **The instantiation of a formula is an assignment of values `true` or `false` to all literals such that the same literals achieve the same value.**

- **A formula is satisfiable if there is an instantiation such that the formula evaluates to true.**
  - The satisfiability problem of propositional logics is always solvable because there are only finitely many combinations in the potential solution space which may be tested successively.
  - Unfortunately, successive testing takes very long time (exponential in the number of literals). Until now no more efficient algorithm is known.

Problem is NP-complete !

Predicate logics (first order)

Predicate logics extends propositional logics by the following:

- **predicates**
  - propositions depending on variables.
    If a proposition depends on k variables, it is called k-ary.
- **variables**
  - correspond to the literals of propositional logics,
    but may assume one out of a set of arbitrarily many values
- **functions**
  - unique assignments depending on variables
    (if a function depends on k variables, it is called k-ary)
  - 0-ary functions are constants.
- **quantors**
  - existence quantor (∃) und all quantor (∀)
  - Quantors must be applied to variables only (otherwise not first order)

Predicate logics (first order)

A predicate logic formula (first order) is built by the following rules:

- **A term is a variable or a k-ary function (using any symbol for the function name)**
- **A formula is a k-ary predicate with arbitrary terms as input or the conjunction, disjunction or negation thereof.**
- **A formula may also contain quantors applied to variables**

Ex.: formula $\varphi = \forall x \, (R(f(y), g(z,y)) \wedge \exists y \, (\neg P(g(y,z), x) \vee R(y, z)))$

Green occurrences of *y* and *z* are **free**.
Red occurences of variables are **bound**.

Closed formulae (constants): Formulae not containing any free variable.

Open formulae (without quantors): Formulae not containing any bound variable.

Atomic formulae: Formulae consisting of one predicate involving terms only (no disjunctions, conjunctions or negations)

Predicate logics (first order)

- **The instantiation of a formula is an assignment of *values to the free variables from predefined domains of definition* such that the same variables achieve the same values.**

- **A formula is satisfiable if there is an instantiation such that the formula evaluates to true.**

☹

  - In predicate logics, the satisfiability problem is not decidable, i.e. no algorithm may ever exist to decide for an arbitrary formula as input if the formula is satisfiable or not.

The general problem is unsolvable !

Is there a work-about ?

Yes, solve a more specific problem !

Power of Prolog

PROLOG does not accept arbitrary predicate formulae:

- ***Domains for variables and functions are arbitrary.***
- ***no quantors***
- ***In CNF, all clauses must be Horn clauses:***

$\neg p \vee \neg q \vee \ldots \vee \neg r \vee x$	*At most one literal is positive*

Rule-based notation of Horn clauses:

$p \wedge q \wedge \ldots \wedge r \rightarrow x$
Rule (Horn clause)

In the assumption there may be a conjunction of positive literals only..

<u>Proposition (Completeness of Horn clause calculus):</u>

For each set of old Horn clauses and a given new Horn clause, Prolog may decide after finite time if the new clause can be concluded from the old clauses ***or not.***

Remark „Finite time" includes „very long" !

Use of Prolog

Didactic use:

- good exercise for dealing with formal logics
- exercising recursive formulations of problems and algorithms

Practical use:

- good for a quick test of concepts (rapid prototyping)
- relatively comfortable for simple problems for which no other solution exists than exhaustive search of all possibilities
- suitable for successive and systematic output of all possible solutions of a search problem

Limits:

- Rather a toy than a tool of commercial use, too far from practical needs
- totally useless if efficiency of solution is relevant