Applications of Artificial Intelligence

Sebastian Iwanowski
FH Wedel

Chapter 2:
Logic and Rule-Based Programming

Features of classical AI solutions

Intelligent creatures are able to process very general knowledge: The more general, the more intelligent.

The ability to process general knowledge needs general description languages for data and processes.

The most general description language is the language of mathematical logics.

This is why traditional AI implementations work with logic description languages.

Problems:

- **The tasks are usually formulated in a different way.**
- **There is a trade-off between generality and efficiency.**

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

Propositional formulae

- **A propositional formula 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 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 be a quantor applying to a variable in a formula**

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

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

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 !

Logic programming languages

Goal:

Design of an algorithm capable to solve a problem formulated in propositional logics.

Due to the previous statement, the problem must belong to a special class !

Advantage:

Logic programming reduces to specification of a problem in propositional logics using a certain set of expressions.

- 🙂 This spares with the design of algorithms for specific problems.
- ☹ The general solution is not adapted to any specific problem and may thus be inefficient.

Key technique to enable a general solution:

😐 ➔ **resolution**

$$\boxed{(p \lor q) \land (r \lor \neg q) \Rightarrow (p \lor r)}$$

generation of a resolvent

- ☹ In the worst case, this is no advantage to pure enumeration of all possibilities.

Logic programming languages

<u>Task for the interpreter:</u>

Original goal: <u>Construction task</u>

less than ever not decidable for arbitrary formulae

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

Easier goal: <u>Verification task</u>

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 !

Logic programming languages

Method 1: Finding a contradiction using *resolution*

Task:

Given a set $\mathcal{F}$ of predicate logic formulae without quantors: Determine whether it is contradictory.

Method:

Try to derive the constant $\perp$ from $\mathcal{F}$ using logic implications.

Principle of resolution:

Generate a new formula being an implication of two given ones:

Principle: Find a literal c ocurring in one formula as $a \vee c$ and in a different formula as $b \vee \neg c$.

Then c may be **eleminated**: $(a \vee c) \wedge (b \vee \neg c) \rightarrow (a \vee b)$

The new formul is called **resolvent** of the old formulae.

Such eliminations may isolate single literals:

Bsp.: $(a \vee c) \wedge \neg c \rightarrow a$ Interpretation: a must hold in $\mathcal{F}$.

If you happen to isolate the negation too, you get a contradiction: ***contradiction!***

Bsp.: $(\neg a \vee d) \wedge \neg d \rightarrow \neg a$ Interpretation: $\neg a$ must hold in $\mathcal{F}$.

Logic programming languages

Method 2: Reduce the variety of terms using *unification*

Example:

$$\Phi = \{\neg P(x, f(y)), P(z, f(g(z)))\}$$

Query: Under which condition may this set of formulae be contradictory ?

Answer: Identify the predicates $P(x, f(y))$ und $P(z, f(g(z)))$ using a proper choice for y and z.

Mere resolution may not discover this!

A logic programming language needs the ability of *unification*:

Replace variables by terms such that two predicates become equal.

Logic programming languages

Method 2: Reduce the variety of terms using *unification*

Substitution:

The substitution [*x*/*t*] applied to φ denotes the formula derived from φ, if all ***free*** occurrences of x in φ are substituted by term *t*.
Analogously, one may define the simultaneous substitution [x1/t1, x2/t2, ..., xn/tn].

Notation: σ = [x1/t1, x2/t2, ..., xn/tn] denotes a ***substitution*** (indepedent of the formula applied to)

$\sigma\,\varphi$ *is the* ***application*** *of substitution* σ to formula φ

Example: formula: φ = P (f (x), y)
substitution: σ = [x/z, y/f (z)]
application: σ φ = P (f (z), f (z))

Definition:

A substitution σ ***unifies*** formulae α_1 und α_2, if the following holds: $\sigma\alpha_1 = \sigma\alpha_2$.

Example:

The predicates *Q* (*f* (*x*), *v*, *b*) and *Q* (*f* (*a*), *g* (*u*), *y*)
are unified by substitution σ = [*x*/*a*,*v*/*g*(*u*), *y*/*b*]

Logic programming languages

Method 2: Reduce the variety of terms using *unification*

Theorem (Existence):

For two predicate expressions there is either a most general unifying substitution which is unique except for renaming variables, or the expressions cannot be unified at all.

Theorem (Computability):

There is an algorithm proving either the non-unifiability of two expressions or finding the most general unifying substitution.

Algorithm for predicate expressions: *rather simple !*

```
Repeat until predicates are equal or non-unifiability is proven:
        If predicate names are different or the number of parameters is different
                → not unifiable
        Else
            Choose a variable x in first or second predicate
                and a term t in the other predicate
                being at the same parameter position and not containing x.
            If this is not possible
                → not unifiable
            Else
                Replace x in both predicates by t.
```

Logic programming languages

Method 2: Reduce the variety of terms using *unification*

Exercises for unification:

$P(x)$ and $Q(y)$

$P(x, y)$ and $P(z)$

$P(x, y)$ and $P(a, f(a))$

$P(x, y)$ and $P(f(z), g(z))$

$P(x, f(x, x), z, f(z, z))$ and $P(f(a, a), y, f(y, y), u)$

$P(x, f(y))$ and $P(z, f(g(z))$

$P(x, x)$ and $P(f(y), f(g(z))$

$P(x, f(x))$ and $P(y, y)$

$P(x, a)$ and $P(b, x)$

The logic programming language PROLOG

Principle of PROLOG:

PROLOG tries to derive a contradiction to a query (considered as a new formula) and a knowledge base (considered as a given formula in conjunctive normal form) using **resolution** and **unification**.

Theorem (completeness with respect to contradictions):

If the system of formulae is contradictory, this may always be discovered.

What is missing ?

Theorem (proving the derivation of a new formula):

If one can find a contradiction to a certain system of formulae,
one can prove for such a system and each new formula ***derivable from that system***,
that the new formula is a derivation from that system.

What is missing here ?

The logic programming language PROLOG

How can we make PROLOG to be <u>complete</u> ?

By restriction to the input !

PROLOG only accepts formulae of the following form:

$\boxed{\mathbf{p \land q \land \ldots \land r \rightarrow x}}$

rules (Horn clauses)

In the proposition, there may only be a set of positive variables.

<u>Theorem (Completeness of resolution for Horn clauses):</u>

☺ For each set of given Horn clauses and a new Horn clause one can decide after finite time whether the new clause is implied from the old clauses ***or not***

☹ ***Remark: „Finite time“ may be „very long“ !***

The logic programming language PROLOG

How can we detect if a set of formulae is eqquivalent to a set of Horn clauses ?

Denote the input formula set in conjunctive normal form. Interpret the clauses to be the individual formulae.
By definition, each formula is a Horn clause, if it is equivalent to a rule where the proposition contains positive variable only.

A Horn clause always looks as follows:

$\neg p \lor \neg q \lor \ldots \lor \neg r \lor x$ *At most one variable is positive.*

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,*
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
$\hookrightarrow$ gibt auch einen Überblick über Prolog und enthält weiterführende Literaturliste

Elements of PROLOG

Elementary components:

- **numbers**

  Integer and real numbers are distinguished (1 ≠ 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: $a \wedge b$

- **implication**

  a :- b corresponds to: $b \rightarrow a$

- **equivalence**

  a = b corresponds to: $b \leftrightarrow a$

- **antiequivalence (exor)**

  a \= b corresponds to: $b \nleftrightarrow a$

- **version-specific operators for comfort**

  not, 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`
  weist der Variable Varname das Ergebnis des arithmetischen Ausdrucks zu.

- **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.

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 a different value for the same clause.
  Shallow Backtracking: Try to achieve a different clause for 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!

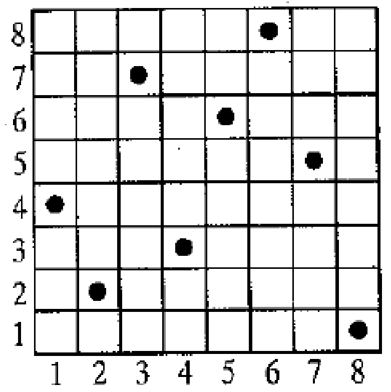
better:

```
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:

```
?-template(S), queens1(S).
```

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