Lecture 8: Introduction to Logic Programming

Logic, Logic Programming, and Prolog

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

What is logic?

- Logic = systematic study of rules to draw conclusions from given assumptions.
- Types of logic
  - Propositional logic
  - First-order logic (FOL): **the most popular type of logic.**

What is logic?

- Logic = systematic study of rules to draw conclusions from given assumptions.
- Types of logic
  - Propositional logic
  - First-order logic (FOL): **the most popular type of logic.**

  FOL allows to do the following:
  1. to represent the objects of interest with terms.
  2. to represent the properties of objects and the relations between them with formulas.
  3. to draw conclusions (theorems) from given assumptions according to a set of rules of deduction.

Terms describe objects of interest. They are built with

- function symbols $f, g, h, \ldots$ from a set F. Every $f \in F$ has an arity $arity(f) \in \mathbb{N}$, and represents either
  - a data constructor, or
  - an operation that produces an object.
- variables $X, Y, Z, \ldots$ from a set of variables $\mathcal{X}$.

using the grammar

$$term ::= X \mid f(term_1, \ldots, term_n) \quad \text{where } X \in \mathcal{X} \text{ and } n = arity(f)$$

- We write $T(F, \mathcal{X})$ for the set of these terms.
- Terms of the form $f()$ for $f \in F$ with $arity(f) = 0$ are called constants. It is customary to write f instead of $f()$

- `car(dacia,color(red),2018)`
  is a term that can describe a Dacia red car built in 2018. It contains the function symbols `car`, `dacia`, `color`, `red`, `2018` with arities 3,0,1,0,0.
  - All these function symbols represent data constructors. `dacia`, `red`, `2018` are constants.
- `+(1,2)`
  is a term that describes the result of adding numbers `1` and `2`. The numbers are represented by constants, and `+` is a function symbol with arity 2 that represents the operation of addition.
  - For some terms, we are allowed to use a more human-readable notation. For example, we prefer to use the infix notation `1+2` instead of `+(1,2)`.

Formulas describe possible relations among objects. They are built with

- terms $t_1, \ldots, t_n \in T(F, \mathcal{X})$ that describe the objects of interest.
- predictate symbols p, r from a set Π. Every $p \in \Pi$ has an arity $arity(p) \in \mathbb{N}$ and is the name of a relation among objects.
- The logical connectives $\vee$ (for disjunction), $\wedge$ (for conjunction), $\neg$ (for negation), $\rightarrow$ (for implication), and $\leftrightarrow$ (for equivalence).
- The quantifiers $\forall$ and $\exists$

using the grammar

$$\begin{aligned} formula \ ::= & \ p(term_1, \ldots, term_n) \\ & \mid \neg formula_1 \mid formula_1 \vee formula_2 \mid formula_1 \wedge formula_2 \\ & \mid formula_1 \rightarrow formula_2 \mid formula_1 \leftrightarrow formula_2 \\ & \mid \forall X.formula_1 \mid \exists X.formula_1 \end{aligned}$$

Formulas $p(t_1, \ldots, t_n)$ are called atomic formulas, or just atoms. If $arity(p) = 0$, we write p instead of $p()$.

- A formula p is an atomic constant.
- The atomic constants `true` and `false` are predefined. `true` represents the always-true formula, and `false` represents the always-false formula.

If $F_1, \ldots, F_n$ are formulas, we write

- $\bigvee_{i=1}^{n} F_i$ instead of $(\cdots(F_1 \vee F_2) \vee \ \ldots) \vee F_n$
  $\bigwedge_{i=1}^{n} F_i$ instead of $(\cdots(F_1 \wedge F_2) \wedge \ \ldots) \wedge F_n$
- For $n = 1$ it is assumed that $\bigvee_{i=1}^{n} F_i = \bigwedge_{i=1}^{n} F_i = F_1$.
- For $n = 0$ it is assumed that $\bigvee_{i=1}^{n} F_i =$ `false` and $\bigwedge_{i=1}^{n} F_i =$ `true`.

A literal is either an atom or the negation of an atom:

$$literal \ ::= \ p(term_1, \ldots, term_n) \mid \neg p(term_1, \ldots, term_n)$$

1. "John is a bright student who likes astronomy."

   $\texttt{student}(\texttt{john}) \wedge \texttt{bright}(\texttt{john}) \wedge \texttt{likes}(\texttt{john}, \texttt{astronomy})$

   - predicate symbols: $\texttt{student}, \texttt{bright}, \texttt{likes}$
   - function symbols: $\texttt{john}, \texttt{astronomy}$

2. "Every human is mortal."

   This sentence has the same meaning as "for all X, if X is a human then X is mortal", and can be translated into

   $\forall \texttt{X}.(\texttt{human}(\texttt{X}) \rightarrow \texttt{mortal}(\texttt{X}))$

3. "Some birds can not fly."

   This sentence has the same meaning as "There is an X such that X is a bird and X can not fly". and can be translated into

   $\exists \texttt{X}.(\texttt{bird}(\texttt{X}) \wedge \neg \texttt{flies}(\texttt{X}))$

Characteristics of first-order logic (I)

First-order logic is also known as Predicate logic or First-order predicate logic. FOL is used often to represent knowledge in AI. It consists of two parts:

1. The language, which provides **terms** to represent objects, and **formulas** to represent knowledge about their properties and the relations that hold among them.
2. Rules of inference, that allow us to derive new knowledge from the knowledge we know. A rule of inference has the form

$$\frac{H_1 \quad \ldots \quad H_n}{C} \quad \text{where } H_1, \ldots, H_n, C \text{ are formulas}$$

with the intended reading "C is a logical consequence (conclusion) of the hypotheses $H_1, \ldots, H_n$".

Characteristics of first-order logic (II)

The language of FOL is an artificial language that can be used to express only sentences to which we can assign a value of truth.

- Sentences like "Who am I?" or “Follow the rules!” can not be expressed in the language of FOL.

Some inference rules specific to FOL

Expansion rule: $\dfrac{A}{B \vee A}$

Cut rule: $\dfrac{A \vee B \quad \neg A \vee C}{B \vee C}$

$\exists$-introduction rule: $\dfrac{A \rightarrow B}{(\exists X.A) \rightarrow B}$

if variable X has no free occurrences in formula B.

First-order logic for Logic Programming

In this course, we are interested in the use of FOL formulas to represent knowledge in Logic Programming with Prolog. The only formulas used in Prolog to represent knowledge are the Horn clauses, that is, formulas of the form

$$\forall X_1 . \cdots \forall X_r . \bigvee_{i=1}^{n} literal_i$$

where at most one literal is an atom, and $X_1, \ldots, X_r$ are all variables that occur in the formula.

There are two kinds of Horn clauses:

- Rules: contain one positive literal: $\forall X_1 . \cdots . \forall X_r . (A \vee \bigvee_{i=1}^{n} \neg B_i)$
- Goals: all literals are negative: $\forall X_1 . \cdots . \forall X_r . \bigvee_{i=1}^{n} \neg B_i$

where $A, B_1, \ldots, B_n$ are atoms.

Rules

Rule = Horn clause with one positive literal:
$\forall X_1. \cdots .\forall X_n. \left(A \vee \bigvee_{i=1}^{n} \neg B_i\right)$
where $A, B_1, \ldots, B_n$ are atoms. This is logically equivalent with

$$\forall X_1. \cdots .\forall X_r. \left(\bigwedge_{i=1}^{n} B_i \rightarrow A\right)$$

- A is the head, and $B_1 \wedge \ldots \wedge B_n$ is the body of the clause.
- A rule whose body is empty (that is, $n = 0$) is a fact.

Rules

Rule = Horn clause with one positive literal:
$\forall X_1. \cdots . \forall X_n. \left(A \vee \bigvee_{i=1}^{n} \neg B_i\right)$
where $A, B_1, \ldots, B_n$ are atoms. This is logically equivalent with

$$\forall X_1. \cdots . \forall X_r. \left(\bigwedge_{i=1}^{n} B_i \rightarrow A\right)$$

- A is the head, and $B_1 \wedge \ldots \wedge B_n$ is the body of the clause.
- A rule whose body is empty (that is, $n = 0$) is a fact.

Interpretations of a rule

Declarative: A holds if B_1 and ... and B_n hold.

Procedural (Kowalski): To solve A, we must solve B_1 and ... and B_n. In this way, the goal of solving A is reduced to the goals of solving B_1 and ... and B_n.

Queries and goals

Query = formula $\exists X_1. \cdots . \exists X_r. \bigwedge_{i=1}^{n} B_i$ where $B_1, \ldots, B_n$ are atoms. Intended reading: "Find $X_1, \ldots, X_r$ such that the formula $B_1 \wedge \ldots\ldots \wedge B_n$ can be deduced from what we know."

- A ground query is a query without variables. The intended reading of $\bigwedge_{i=1}^{n} B_i$ is: "Check if the formula $B_1 \wedge \ldots\ldots \wedge B_n$ can be deduced from what we know."

Goal = negation of a query: $\neg\exists X_1. \cdots . \exists X_r. \bigwedge_{i=1}^{n} B_i$.

Remark

A goal is logically equivalent with $\forall X_1. \cdots . \forall X_n. \bigvee_{i=1}^{n} \neg B_i$.

- This is a Horn clause there all literals are negative.

What is Logic Programming?

Logic Programming = declarative programming style where

- **Knowledge** is encoded as a collection of rules and facts collected in a program P.
- **Computation** is triggered by running a **query**

$$\exists X_1. \cdots .\exists X_r. \bigwedge_{i=1}^{n} B_i$$

and is solved by a fixed and predictable strategy, called SLDNF-resolution. An answer is a substitution $\theta = [term_1/X_1, \ldots, term_r/X_r]$ such that, if $B'_i = \theta B_i$ for $1 \leq i \leq r$ then $\bigwedge_{i=1}^{n} B'_i$ can be deduced logically from P.

  - We write $P \vdash formula$ to indicate that a formula *formula* can be deduced logically from program P

What is Prolog?

Prolog: the main language for Logic Programming.

- Developed and implemented by A. Colmerauer and P. Rousell, in 1972. It is based on the procedural interpretation of Horn clauses.

Applications:

theorem proving, expert systems, term rewriting, type systems, automated planning, natural language processing.

Implementations: SICStus Prolog, Ciao, Visual Prolog, SWI-Prolog

We will practice logic programming with SWI-Prolog.

- Cross-platform, freely available from here.

There is only one data type in Prolog: terms. Terms describe objects, and are either

- **Atoms**: general-purpose names without inherent meaning: `x`, `red`, and `'some atom'`. They should start with lowercase letter, or be delimited by quotes.
- **Numbers**, which can be can be floats or integers. SWI-Prolog supports working arbitrary-length integers.
- **Strings**: `"to be, or not to be"`, `""`
- **Variables** are placeholders for arbitrary terms. They are represented by strings made of letters, numbers and underscore characters, and beginning with an uppercase letter or underscore. Examples: `X`, `Y`, `_X`
- **Compound terms** of the form $f(term_1, \ldots, term_n)$ where f is an atom with arity $n > 0$. f is called a functor.

- A list is an ordered collection of terms [$term_1$,...,$term_n$].
  - Examples: `[]` (empty list), `[red,green,blue]`
  - Other notations for [$term_1$,...,$term_n$] are:

$$[\underbrace{term_1}_{\text{head}} \mid \underbrace{[term_2, \ldots, term_n]}_{\text{tail}}] \text{ or}$$

$$[\underbrace{term_1, \ldots, term_k}_{\text{first } k \text{ terms}} \mid [term_{k+1}, \ldots, term_n]]$$

- The following predicates are predefined for type-checking:

```
atom
number
string
integer
float
```

Syntax of Prolog

Program = collection of rules and facts

In Prolog, a **rule** $\forall X_1. \cdots .\forall X_r.(\bigwedge_{i=1}^{n} B_i \rightarrow A)$ is written as

$$A \ :- \ B_1, \ldots, B_n .$$

with the intended reading "A if B_1 and ... and B_n".
A **fact** $\forall X_1. \cdots .\forall X_r.A$ is written as

$$A.$$

with the intended reading "A holds".

Remarks

1. Note the mandatory parts in the syntax of rules: '$:-$', '$,$' and '$.$'.
2. The universal quantifiers for all variables are implicit.
3. The head A of a rule is always an atom $p(term_1, \ldots, term_k)$ where p is a predicate symbol: A rule with such a head is *a defining rule for the predicate p*.

Program (Prolog) = text file with defining rules and facts. It should have extension `.pl`
For example:

```
% father(X,Y) means that X is the father of Y.
% mother(X,Y) means that X is the mother of Y.
father(john,jack).          % John is father of Jack (fact)
father(john,bob).           % John is father of Bob  (fact)
mother(mary,jack).          % Mary is mother of Jack (fact)
mother(ana,ray).            % Ana is mother of Bob  (fact)
% parent(X,Y) holds if X is a parent of Y.
parent(X,Y):-father(X,Y).   % defining rule 1 for parent
parent(X,Y):-mother(X,Y).   % defining rule 2 for parent
% siblings(X,Y) holds if X and Y are different terms and have a common parent Z.
siblings(X,Y):-             % defining rule for siblings
  parent(Z,X),
  parent(Z,Y),
  X\=Y.
```

Program (Prolog) = text file with defining rules and facts. It should have extension `.pl`
For example:

```
% father(X,Y) means that X is the father of Y.
% mother(X,Y) means that X is the mother of Y.
father(john,jack).              % John is father of Jack (fact)
father(john,bob).               % John is father of Bob  (fact)
mother(mary,jack).              % Mary is mother of Jack (fact)
mother(ana,ray).                % Ana is mother of Bob  (fact)
% parent(X,Y) holds if X is a parent of Y.
parent(X,Y):-father(X,Y).   % defining rule 1 for parent
parent(X,Y):-mother(X,Y).   % defining rule 2 for parent
% siblings(X,Y) holds if X and Y are different terms and have a common parent Z.
siblings(X,Y):-                 % defining rule for siblings
  parent(Z,X),
  parent(Z,Y),
  X\=Y.
```

REMARK: the text after `%` is comment for humans to read. It is ignored by Prolog.

Syntax of Prolog

More about predicates

The predicates defined in programs are user-defined predicates:

- They are the names p that occur in heads $p(term_1, \ldots, term_n)$ of rules and facts. Their meaning is defined by the program rules.
- Examples: `father`, `mother`, `parent`, `sibling` in the program illustrated before.

Prolog also has predefined predicates:

- The type-checking predicates `atom`, `number`, `string`, `integer`, `float`
- Predicates to test equality: $term_1 = term_2$ and disequality: $term_1 \backslash= term_2$
- Comparison predicates: `<`, `>`, `>=` and `=<` (less or equal).
- The predicate `is` which enforces the evaluation of terms with predefined functions (see next slides).
- ...

Preliminary remarks about Prolog

FP versus LP

In **functional programing** (FP), programs consist of function definitions

- There are two kinds of functions: user-defined and predefined
- Some functional languages (e.g., Haskell) also allow us to define type classes and datatypes.

In **logic programming** (LP), programs consist of predicate definitions

- There are two kinds of predicates: user-defined and predefined
- Prolog recognizes some built-in operations too: arithmetic operators `+`, `-`, `*`, `/`, trigonometric functions, etc.
  - Terms with function calls are not evaluated (in FP they are evaluated automatically), but Prolog has the built-in predicate `is` to enforce the evaluation of terms with predefined functions (see next slides).

Start SWI-Prolog by double-clicking its icon 

$\Rightarrow$ An interactive window will open, where users can type queries after the ?– prompt.

Queries are formulas $\exists X_1 . \cdots . \exists X_r . \bigwedge_{i=1}^{n} A_i$ where A_i are atoms. Their intended reading is: "Find the values of $X_1, \ldots, X_r$ for which the formula $A_1 \wedge \ldots \wedge A_n$ follows from what we know." If the formula has no variables, the intended reading is simpler: "Check if $A_1 \wedge \ldots \wedge A_n$ follows from what we know".

- In SWI-Prolog, such a query is written

```
?- A1, ..., An .
```

after the ?– prompt. All variables (if any) are existentially quantified by default. Prolog uses a search strategy called **SLDNF-resolution** to find the answers to the query.

The knowledge base used by Prolog to find answers to queries consists of

1. The predefined predicates.
2. The definitions of predicates defined in the programs consulted by the user via the `File` menu.

   REMARK. The `File` menu allows users to create, open, edit, save and consult program files.

Example (Queries with predefined predicates)

```
?- f(X,a)=f(b,Y).
X = b,
Y = a.
?- X=1+2*3,Y=f(X).
X = 1+2*3,
Y = f(1+2*3).
?- [_,Y|Z]=[a,b,c,d].
Y = b,
Z = [c, d].
```

```
?- f(X,a)=f(b,X).
false.

?- X is 1+2*3,Y=f(X).
X = 7,
Y = f(7).
?- [X,Y|_]=[1,2,3],X>Y.
false.
```

The atom $term_1 = term_2$ computes the most general unifier $[\theta]$ that instantiates the variables in $term_1$ and $term_2$ with terms. If θ exists, it instantiated the variables and the atom holds; otherwise the atom does not hold.

- $term_1$ are $term_2$ are not evaluated.
- _ is an anonymous variable (like in Haskell).

The variables used in LP are called logical variables. They can be uninstantiated or instantiated with a term.

> The value of a logical variable can not be changed, but can be further instantiated.

The atom `X is` *term* holds if `X` is an uninstantiated variable and *term* is term for an arithmetic expression whose value is a number. In this case, Prolog computes the numeric value *v* of *term* and `X` is instantiated with *v*.

Working with user-defined predicates

We can use of the knowledge about `mother`, `father`, `parent` and `siblings` if we consult the program from file `Prog1.pl` with the content illustrated before (download from here):

Examples

```
?- siblings(X,Y).
X = jack,
Y = bob ;
X = bob,
Y = jack ;
false.
```

```
?- parent(X,jack).
X = john ;
X = mary.
```

The query "Find `X`, `Y` who are siblings" has two answers: `X=jack`, `Y=bob` and `X=bob`, `Y=jack`

The query "Find `X` who is parent of `jack`" has two answers: `X=john` and `X=mary`