Logic & Prolog

Roberta Calegari

roberta.calegari@unibo.it

Alma AI - Alma Mater Research Institute for Human-Centered Artificial Intelligence—Università di Bologna

Master in diritto delle nuove tecnologie e informatica giuridica
Academic Year 2020/2021

1. Logic Programming
2. Prolog

Next in Line...

Origins I

Early history [Apt, 2005]

- automatic deduction of theorems
- *first-order logic* (FOL) by Frege, Peano and Russell
- computation as deduction by Gödel and Herbrand
- *resolution* principle by Robinson [Robinson, 1965], along with *unification*

The key issue

- resolution by Robinson
  - allowed proof of FOL theorem made it possible to compute with logic
  - not yet to see logic as a full computational framework
- from computable logic to logic as a programming language something was still missing

Origins II

The procedural interpretation of Horn clauses

- by defining logic programs as collections of Horn clauses
- by restricting Robinson's principle accordingly
- Kowalski showed how a logical implication could be amenable of both a *declarative* and a *procedural* interpretation [Kowalski, 1974]
- thus providing the *foundations* for a logic programming language
- Prolog, by Colmerauer in Marseille, came along in 1973

There is no question that Prolog is essentially a theorem prover à la Robinson. Our contribution was to transform that theorem prover into a programming language. [Colmerauer and Roussel, 1996]

Essentials I

Three fundamental features [Apt, 2005]

terms *Computing takes place over the domain of all terms defined over a "universal" alphabet.*

mgu *Values are assigned to variables by means of automatically-generated substitutions, called most general unifiers. These values may contain variables, called logical variables.*

backtracking *The control is provided by a single mechanism: automatic backtracking.*

Basic Units of Computation I

Atomic actions [Apt, 2005]

- logic programming is a different paradigm for programming languages
- since it is ruled by different *principles* w.r.t. the other sorts of programming languages
  - atomic actions are *equations* between *terms*
  - executed by means of the *unification* process trying to solve them
  - unification assigns *values* to *variables*
  - values can be *arbitrary terms*—in fact, there is just one sort of variable, ranging over the set of all terms
- so, in order to understand logic programming as a computational paradigm, we first need to understand its basic units of computation

Basic Units of Computation II

Terms: Definition

- a *variable* is a *term*
- a *functor* (or, *function symbol*) with arity 0 is called a *constant*, and is a term
- if f is a functor of arity n, and $t_1, \ldots, t_n$ are n terms, then $f(t_1, \ldots, t_n)$ is a term

Basic Units of Computation III

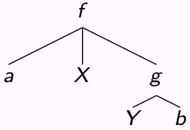
Terms: Examples

- let's say that X, Y are variables, a, b constants (or, functors of arity 0), f, g functors of arity $3, 2$ respectively. Then
  - a, b, X, and Y are proper terms
  - $f(a, b, a)$ and $g(X, Y)$ are proper terms
  - $f(a, X, g(Y, b))$ is a proper term
- variables and constant are *atomic terms*, terms built out of proper functors are *structured terms*. Then
  - a, b, X, and Y are atomic terms
  - $f(a, b, a)$, $g(X, Y)$ and $f(a, X, g(Y, b))$ are structured terms
  - ! in the structured term $f(a, X, g(Y, b))$, f is the *functor symbol* of *arity* 3, whereas $a, X, g(Y, b)$ are the three *subterms*

Basic Units of Computation IV

Terms: Remarks

- a *recursive* definition, leading to a recursive data structure—a tree
  - e.g., structured term $f(a, X, g(Y, b))$ maps onto tree



- fundamental in mathematical logic, terms are *essential in computer science*, too: e.g., they capture both arithmetic expressions and strings
- no specific alphabet is assumed—*universal alphabet* for all terms
- no meaning is a *a priori* attached to symbols, in particular to functors—e.g., $+$ is just a functor, not associated a priori with the plus sign of arithmetic

→ no types

Logic Programs I

Logic clause

- a logic clause is a (finite) disjunction of literals [Console et al., 1997]
- if $A_1, \ldots, A_n, B_1 \ldots, B_m$ are atoms, containing variables $X_1, \ldots, X_k$, then

$$\forall X_1, \ldots, X_k (A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m)$$

  is a logic clause, which is *logically equivalent* to

$$\forall X_1, \ldots, X_k ((A_1 \vee \ldots \vee A_n) \leftarrow (B_1 \wedge \ldots \wedge B_m))$$

  usually written simply as

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

- a clausal normal form (CNF) is a *conjunction of clauses*

Logic Programs II

Definite clauses

- a definite clause, has just one positive literal ($n = 1$)

$$A \leftarrow B_1, \ldots, B_m$$

- a unitary clause, is a definite clause with no negative literal ($m = 0, n = 1$)

$$A \leftarrow$$

- a definite goal is a definite clause with no positive literal ($n = 0$)

$$\leftarrow B_1, \ldots, B_m$$

Horn clauses

- a Horn clause is either a definite clause or a definite goal ($n = 1$ or $n = 0$)

Logic Programs III

Logic program

- in a logic program
  - a definite clause is called a rule
  - a unitary clause is a fact
  - a definite goal is just a goal
- a logic program is a CNF of Horn clauses
  - so, it is a conjunction of rules and facts (and goals)

Goals & Proofs I

Proving goals

- Robinson's principle proceed by contradiction, trying to prove a formula F false against CNF H, succeeding if this fails
  - technically, proving that $H \cup \neg F$ is not satisfiable
- proving an atom G in logic programming amounts at proving $\neg G$ against logic program P
  - technically, proving goal $\leftarrow G$ on P
- computation in logic programming proceeds by *proving goals*

! resolution leads to *backward chaining*—from goal back to axioms

Goals & Proofs II

SLD resolution informally

- to prove a goal G w.r.t. program P, the resolution principle for logic programming proceeds according to the procedural interpretation
- so, first we look for one clause $A \leftarrow B_1, \ldots, B_n$ in P whose head A unifies with G
- if the most general unifier of G and A is θ ($mgu(G, A) = \theta$), then the proof of G succeeds if we can further prove $B_1\theta, \ldots, B_n\theta$—where $B_i\theta$ represents the application of the *mgu* θ to B_i

! the application of θ to clause $A \leftarrow B_1, \ldots, B_n$ specialises the clause to the specific atom we need to proof—that is, our current goal

! resolution proceed recursively with the proof of *subgoals* $B_1\theta, \ldots, B_n\theta$

→ in general, the computational state of the SLD resolution include a (possibly empty) conjunction of atom (goals) $G_1, \ldots, G_n$ to be proven—the *current goal* of the proof

An Example I

A simple logic program

$parent(joey, luca)$
$parent(joey, simone)$
$parent(lino, joey)$
$parent(mirella, joey)$

$grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z)$

An Example II

Declarative interpretation

- four *facts* are expressed by means of predicate $parent/2$
  - four propositions that are considered true with no need of proof—our axioms
  - a possible interpretation is that, e.g., *joey* is a parent of *luca*—just one of the many, even though the most intuitive for English speakers
- one *rule* is expressed by means of predicate $grandparent/2$
  - since it is the short form for

    $$\forall X, Y, Z, grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z)$$

    it means that formula $grandparent(X, Z)$ holds if both $parent(X, Y)$ and $parent(Y, Z)$ are true, whatever the values of X, Y, Z
  - so, it can be used to prove the truth of, e.g., formula $grandparent(lino, luca)$ since both $parent(joey, luca)$ and $parent(lino, joey)$ are true since they are facts in the logic program
  - independently of the possible interpretations

An Example III

Procedural interpretation

- two procedure are defined: *parent*/2 and *grandparent*/2
- two (procedure) calls can be executed correspondingly—goals of the form
  - $\leftarrow parent(?, ?)$
  - $\leftarrow grandparent(?, ?)$

  with any sort of term in the place of the ?
  - for instance, $\leftarrow grandparent(lino, luca)$
- to compute *parent*/2 we can use the four facts, non-deterministically
- to compute *grandparent*/2 we can use the rule, first matching the rule head, then proceeding by calling the two subprocedures, via the two *subgoals* of the form *parent*/2
  - for instance, to compute $\leftarrow grandparent(lino, luca)$ we will compute subgoals $\leftarrow parent(lino, Y)$ and $\leftarrow parent(Y, luca)$

An Example IV

Possible goals

- $grandparent(lino, luca)$ succeeds—one refutation, no *computed substitution*
- $grandparent(lino, joey)$ fails—no refutations
- $grandparent(lino, X)$ succeeds twice—two refutations, two different computed substitutions
  - $X/luca$
  - $X/simone$
- $grandparent(X, simone)$ succeeds twice—two refutations, two different computed substitutions
  - $X/lino$
  - $X/mirella$
- $grandparent(X, Y)$ succeeds four times—four refutations, four different computed substitutions
  - $X/lino$, $Y/luca$
  - $X/lino$, $Y/simone$
  - $X/mirella$, $Y/luca$
  - $X/mirella$, $Y/simone$

An Example: Remarks

Remarks

From the example we get some early hints about some benefits of logic programming

- multiple uses of the single program
  - the simple program above can be used to test the family relations between known people, or, to compute them
  - mostly, input / output parameters needs not to be defined a priori
- knowledge-based programming
  - arbitrarily complex relations expressed as FOL facts represent the core of a logic program
  - knowledge representation is straightforward in the logic programming formalism—with FOL
- language for rule-based systems
  - classical AI, such as expert systems [Buchanan and Shortliffe, 1984]

Next in Line. . .

1. Logic Programming

2. Prolog

Prolog examples: facts, atom & predicate I

Imagine we want to encode characters of Homer's Iliad and Odyssey.
→ This translates into Prolog *facts* ended with a period

```
character(priam, iliad).
character(hecuba, iliad).
character(achilles, iliad).
character(agamemnon, iliad).
character(patroclus, iliad).
character(hector, iliad).
character(andromache, iliad).
character(rhesus, iliad).
character(ulysses, iliad).
character(menelaus, iliad).
character(helen, iliad).
```

```
character(ulysses, odyssey).
character(penelope, odyssey).
character(telemachus, odyssey).
character(laertes, odyssey).
character(nestor, odyssey).
character(menelaus, odyssey).
character(helen, odyssey).
character(hermione, odyssey).
```

Prolog examples: facts, atom & predicate II

Fact

Facts are statements that describe object properties or relations between objects.

Knowledge

Such a collection of facts, and later, of rules, makes up a database. It transcribes the knowledge of a particular situation into a logical format. Adding more facts to the database, we express other properties

Prolog examples: facts, atom & predicate III

such as the gender of characters:

```
% Male characters
male(priam).
male(achilles).
male(agamemnon).
male(patroclus).
male(hector).
male(rhesus).
male(ulysses).
male(menelaus).
male(telemachus).
male(laertes).
male(nestor).
```

```
% Female characters
female(hecuba).
female(andromache).
female(helen).
female(penelope).
```

Prolog examples: facts, atom & predicate IV

or relationships between characters such as parentage:

```
% Fathers
father(priam, hector).
father(laertes, ulysses).
father(atreus, menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).
```

```
% Mothers
mother(hecuba, hector).
mother(penelope, telemachus).
mother(helen, hermione).
```

Prolog examples: facts, atom & predicate V

Finally, if we want to describe kings of some cities and their parties, this would be done as:

```
king(ulysses, ithaca, achaean).
king(menelaus, sparta, achaean).
king(nestor, pylos, achaean).
king(agamemnon, argos, achaean).
king(priam, troy, trojan).
king(rhesus, thrace, trojan).
```

Prolog examples: facts, atom & predicate VI

General form of a Prolog fact

$$relation(object_1, object_2, ..., object_n).$$

Symbols or names representing objects → `ulysses` or `penelope`: *atoms*

Atoms

- strings of letters, digits, underscores begin with a lowercase letter
- can also be a string beginning with an uppercase letter or including white spaces, but it must be enclosed between quotes → `'Ulysses'` or `'Pallas Athena'` are legal atoms

Prolog examples: facts, atom & predicate VII

Predicate

- name of the symbolic relation is the *predicate*,
- the objects $object_1, object_2, ..., object_n$ involved in the relation are the *arguments*
- the number n of the arguments is the *arity*

Traditionally, a Prolog predicate is indicated by its name and arity

`predicate/arity`

→ e.g., `character/2`, `king/3`

Prolog examples: terms I

Terms

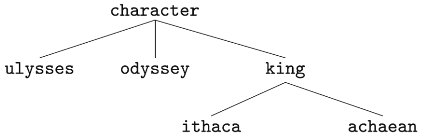
- all forms of data are called *terms*
- constants, i.e., atoms or numbers are terms
- facts, like `king(menelaus, sparta, achaean)`, are a *compound term* or a structure, that is, a term composed of other terms (called subterms)

→ arguments of this compound term are constants

→ can also be other compound terms, as in `character(priam, iliad, king(troy, trojan))` where the arguments of the predicate character/3 are two atoms and a compound term

Prolog examples: tree of terms

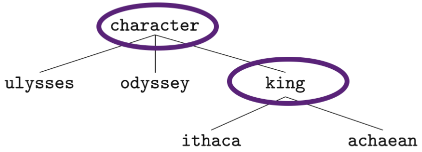
```
character(ulysses, odyssey, king(ithaca, achaean))
```



- nodes of the tree are equivalent to the functors of a term

Prolog examples: tree of terms

```
character(ulysses, odyssey, king(ithaca, achaean))
```



- use trees to represent compound terms
- nodes of the tree are equivalent to the functors of a term

Prolog examples: compound terms

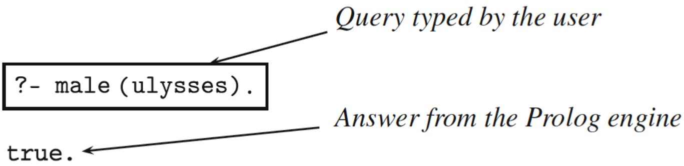
Compound Term

- functor – the name of the relation – and arguments
- leftmost functor of a term is the principal functor
- same principal functor with a different arity corresponds to different predicates: `character/3` is thus different from `character/2`
- constant is a special case of compound term with no arguments and an arity of 0 (can be referred to as `abc/0`)

Prolog examples: query I

Query

- request to prove or retrieve information from the knowledge base
- Prolog answers yes if it can prove it, that is, here if the fact is in the database, or no if it cannot: if the fact is absent (case of asking for a fact tu be proven)
- question *Is Ulysses a male?*



Prolog examples: query II

```
?- male(penelope).
false.
```

- expressions `male(ulysses)` or `male(penelope)` are *goals* to prove
- some questions require more goals, such as *Is Menelaus a male and is he the king of Sparta and an Achaean?*

```
?- male(menelaus), king(menelaus, sparta, achaean).
true.
```

- where "," is the conjunction operator → indicates that Prolog has to prove both goals
- compound queries: conjunction of two or more goals
  `?- G1, G2, G3, ..., Gn.`
- Prolog proves that all the goals $G_1 ... G_n$ are true

Prolog examples: logical variables I

Logical variables

- begin with an uppercase letter, for example, `X`, `Xyz`, or an underscore
- stand for any term: constants, compound terms, and other variables
- term containing variables such as `character(X, Y)` can *unify* with a compatible fact, such as `character(penelope, odyssey)`, with the *substitutions* `X = penelope` and `Y = odyssey`
- Prolog resolution algorithm searches terms in the database that unify with it → substitutes the variables to the matching arguments
- question *What are the characters of the Odyssey?*

Prolog examples: logical variables II

The variable *The query*

```
?- character(X, odyssey).
```

The Prolog answer

```
X = ulysses
```

Prolog examples: logical variables III

- question *What is the city and the party of king Menelaus?*

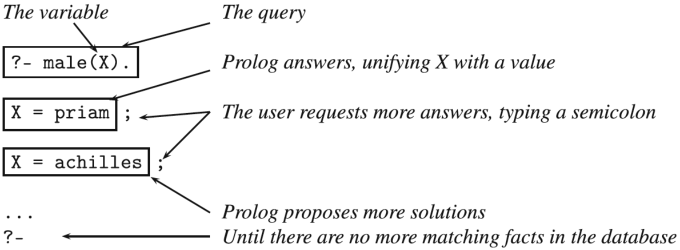
```
?- king(menelaus, X, Y).
X = sparta, Y = achaean

?- character(menelaus, X, king(Y, Z)).
X = iliad, Y = sparta, Z = achaean

?- character(menelaus, X, Y).
X = iliad, Y = king(sparta, achaean)
```

- multiple solutions → Prolog considers the first fact to match the query in the knowledg
- type ";" to get the next answers until there is no more solution

Prolog examples: logical variables IV



Prolog examples: shared variables I

Shared variables

- goals in a conjunctive query can share variables
- constrain arguments of different goals to have the same value
- question *Is the king of Ithaca also a father?*
- conjunction of two goals `king(X, ithaca, Y)` and `father(X, Z)`, with `X` shared between the goals

```
?- king(X, ithaca, Y), father(X, Z).
X = ulysses, Y = achaean, Z = telemachus
```

Prolog examples: shared variables II

- not interested in the name of the child although Prolog responds with `Z = telemachus`.
- can indicate we do not need to know the values of Y and Z using anonymous variables

```
?- king(X, ithaca, _), father(X, _).
X = ulysses
```

Prolog examples: rules I

Rules

- enable to derive a new property or relation from a set of existing ones
- a term called the *head* or *consequent* followed by symbol `:-` (read if) and a conjunction of goals called *antecedent* or *body*

$$HEAD :- G_1, G_2, G_3, \ldots G_n.$$

- the head is true if the body is true
- variables are shared between the body and the head

Prolog examples: rules II

```
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).

?- son(telemachus, Y).
Y = ulysses;
Y = penelope;
```

Rules

Flexible way to *deduce* new information from a set of facts.

The `parent/2` predicate is another example of a family relationship that is easy to define using rules. Somebody is a parent if s/he is either a mother or a father:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Prolog examples: rules III

- Rules can call other rules as with `grandparent/2`
- Z is an intermediate variable shared between goals. $\rightarrow$ enables to find the link between the grandparent and the grandchild: a mother or a father

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

- generalize the `grandparent/2` predicate and write `ancestor/2`
- two rules, one of them being recursive

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Prolog examples: rules IV

- recursion pattern is quite common for Prolog rules
- one or more rules express a general case using recursion
- another set of rules or facts describes simpler conditions without recursion $\rightarrow$ correspond to boundary cases and enable the recursion to terminate

Prolog clauses

Facts and rules are also called *clauses*

Prolog Program

Prolog program

program	a sequence of Prolog clauses
	interpreted as a *conjunction* of clauses
logic theory	constituting a *logic theory* made of Horn clauses written according the Prolog syntax

Prolog Execution I

Aim of a Prolog computation

- given a Prolog program P and the goal `?- p(t1,t2,...,tm)` (also called *query*)
- if `X1,X2,...,Xn` are the variables in terms `t1,t2,...,tm`
- the meaning of the goal is to query P and find whether there are some values for `X1,X2,...,Xn` that make `p(t1,t2,...,tm)` true

→ thus, the aim of the Prolog computation is to find a substitution $\sigma =$`X1/s1,X2/s2,...,Xn/sn` such that $P \models \mathtt{p}(\mathtt{t1}, \mathtt{t2}, \ldots, \mathtt{tm})\sigma$

Prolog Execution II

Prolog search strategy

- as a logic programming language, Prolog adopts SLD resolution
- as a search strategy, Prolog applies resolution in a strictly linear fashion
  - *goals* are replaced *left-to-right*, sequentially
  - *clauses* are considered in *top-to-bottom* order
  - *subgoals* are considered *immediately* once set up

→ resulting in a *depth-first* search strategy

Prolog Execution III

Prolog backtracking

- in order to achieve completeness, Prolog saves choicepoints for any possible alternative still to be explored
- and goes back to the nearest choice point available in case of failure
- exploiting automatic backtracking

References I

Apt, K. R. (2005).
The logic programming paradigm and Prolog.
In Mitchell, J. C., editor, *Concepts in Programming Languages*, chapter 15, pages 475–508. Cambridge University Press, Cambridge, UK.

Buchanan, B. G. and Shortliffe, E. H. (1984).
Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.
The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc.

Calegari, R., Denti, E., Mariani, S., and Omicini, A. (2018).
Logic programming as a service.
Theory and Practice of Logic Programming, 18(5-6):846–873.
Special Issue "Past and Present (and Future) of Parallel and Distributed Computation in (Constraint) Logic Programming".

Colmerauer, A. and Roussel, P. (1996).
The birth of Prolog.
In Bergin Jr., T. J. and Gibson Jr., R. G., editors, *History of programming languages—II*, volume II, pages 331–367. ACM, New York, NY, USA.

References II

Console, L., Lamma, E., Mello, P., and Milano, M. (1997).
Programmazione logica e Prolog.
UTET Libreria.

Denti, E., Omicini, A., and Ricci, A. (2001).
tuProlog: A light-weight Prolog for Internet applications and infrastructures.
In Ramakrishnan, I., editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg.
3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

Denti, E., Omicini, A., and Ricci, A. (2005).
Multi-paradigm Java-Prolog integration in tuProlog.
Science of Computer Programming, 57(2):217–250.

Kowalski, R. A. (1974).
Predicate logic as programming language.
In *Information Processing 74 – Proceedings of the 1974 IFIP Congress*, pages 569–574.
North-Holland Publishing Company.

References III

Martelli, A. and Montanari, U. (1982).
An efficient unification algorithm.
ACM Transactions on Programming Languages and Systems, 4(2):258–282.

Metakides, G. and Nerode, A. (1996).
Principles of Logic and Logic Programming, volume 13 of *Studies in Computer Science and Artificial Intelligence*.
Elsevier.

Nilsson, U. and Maluszynski, J. (1995).
Logic, Programming and Prolog.
John Wiley & Sons, Inc., New York, NY, USA.

Rijk, L. M. D. (2002).
Aristotle: Semantics and Ontology. Volume I: General Introduction. The Works on Logic, volume 91 of *Philosophia Antiqua*.
Brill Academic Publishers.

Robinson, J. A. (1965).
A machine-oriented logic based on the resolution principle.
Journal of the ACM, 12(1):23–41.

Logic & Prolog

Roberta Calegari

roberta.calegari@unibo.it

Alma AI - Alma Mater Research Institute for Human-Centered Artificial Intelligence—Università di Bologna

Master in diritto delle nuove tecnologie e informatica giuridica
Academic Year 2020/2021