Logic Programming in Prolog with tuProlog

Autonomous Systems / Technologies
Sistemi Autonomi / Tecnologie

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2016/2017
1. Lab Activity 1

2. Prolog

3. Lab Activity 2
Next in Line...
Example 1: The parent.pl Knowledge Base I

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

A logic theory
- a simple logic *program*
- with four *ground facts*
- representing one sort of relation between elements of the *domain of discourse*

? is there anything we can do with this program?
?? can we *compute* anything?
Example 1: The parent.pl Knowledge Base II

Constants & predicates

- joey, luca, simone, lino and mirella are *constant* used in the program as *ground terms* to denote the element of the domain.

- parent is the *predicate* used in the program to talk about the domain of discourse—parent/2 says that parent is the *predicate symbol* with *arity* 2.
Example 1: The parent.pl Knowledge Base III

Goals

- since the only predicate in the program is parent/2, we cannot prove anything else, in principle—except for tautologies, or built-in Prolog predicates
- possible goals

1. 
   :- parent(joey,luca).
2. 
   :- parent(joey,lino).
3. 
   :- parent(joey,X).
4. 
   :- parent(X,joey).
5. 
   :- parent(X,Y).

Let us try the above queries in tuProlog
What is tuProlog?

- tuProlog is a light-weight Prolog system for distributed applications and infrastructures [Denti et al., 2001]
- intentionally designed around a minimal core
- to be either statically or dynamically configured by loading/unloading libraries of predicates
- tuProlog natively supports multi-paradigm programming [Denti et al., 2005], providing a clean, seamless integration model between Prolog and mainstream object-oriented languages
tuProlog in Short II

Where is tuProlog?

UniBo  http://tuprolog.unibo.it

BitBucket  http://bitbucket.org/tuprologteam/tuprolog

downloads http://bitbucket.org/tuprologteam/tuprolog/downloads
tuProlog in Short III

What to download?

From http://bitbucket.org/tuprologteam/tuprolog/downloads

v. 3.2.1 http://bitbucket.org/tuprologteam/tuprolog/downloads/2p-3.2.1.zip


Let us try the above queries in tuProlog
Starting tuProlog CLI

- after downloading the tuProlog ZIP file, unzip it, and position your command line in the 2p-3.2.1/bin/ directory
- then invoke the tuProlog command line interface (CLI) with
  
  ```java
  java -cp 2p.jar alice.tuprologx.ide.CUIConsole
  ```
- to see if everything is working, invoke query

  ```prolog
  ?- write('Hello, World!').
  ```
- and see what happens
first, load a simple program with the query

?- set_theory('parent(joey,luca). parent(joey,simone).
parent(lino,joey). parent(mirella,joey).').

then try the following queries:

1. ?- parent(joey,luca).
2. ?- parent(joey,lino).
3. ?- parent(joey,X).
4. ?- parent(X,joey).
5. ?- parent(X,Y).

and see what happens, by responding with

- \langle return\rangle to accept the answer
- ; to have more answers
Remarks on interaction

1. success
2. failure
3. computed substitution
4. unification
5. backtracking

All no input / output parameters: no direction required for arguments in principle thanks to unification
Starting tuProlog GUI

- in the same 2p-3.2.1/bin/ directory as before, invoke the tuProlog graphical user interface (GUI) with:
  
  ```
  java -cp 2p.jar alice.tuprologx.ide.GUILauncher
  ```

- to see if everything is working, invoke query
  ```
  ?- write('Hello, World!').
  ```
  and see what happens

- redo `parent.pl` experiments with GUI
  - load `parent.pl`
  - retry the queries discussed in the CLI experiments
Example 2: grandparent.pl

Adding a rule

- load grandparent.pl
- there, rule

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

  is added to the knowledge base

- now the logic program is a collection of facts and rules

  it is a so-called universal rule

Test the program

- now try the following queries:
  1. ?- grandparent(lino,luca).
  2. ?- grandparent(lino,joey).
  3. ?- grandparent(joey,X).
  4. ?- grandparent(lino,X).
  5. ?- grandparent(X,Y).
Adding another rule

- load sibling.pl
- there, rule
  
  sibling(Y,Z) :- parent(X,Y), parent(X,Z).

  is added to the previous logic theory

- all the previous theorems are true: all previous computations are the same
- just adding new theorems based on a new rule

! operator \=/2 represent an explicit computation over terms
  
  succeeding when the two arguments are terms that do not unify
  
  all other computations over terms till now were implicitly driven by goal unification
Example 3: sibling.pl II

Test the program

- now try the following queries:
  1. ?- sibling(simone,luca).
  2. ?- sibling(lino,joey).
  3. ?- sibling(luca,X).
  4. ?- sibling(lino,X).
  5. ?- sibling(X,Y).

and discuss all the results
Next in Line...
Prolog terms

- **variables**: alphanumeric strings starting with either an *uppercase* letter or an *underscore*
  - underscore alone is the *anonymous variable*—sort of *don’t care* variable
  - underscore followed by a string is a normal variable during resolution, but it does not need to be exposed in the computed substitution

- **functors**: alphanumeric strings starting with a *lowercase* letter
  - holds for both proper functors and constants

- **terms**: are built recursively out of functors and variables as in logic programming
  - e.g., `term, Var, f(X), p(Y,f(a))` are *Prolog terms*
  - e.g., `term, f(a), p(x,y)` are *Prolog ground terms*
Prolog Syntax II

Prolog atoms

- **predicates** alphanumeric strings starting with a `lowercase` letter
  - the same as functors

- **atoms** are built applying predicates to terms as in logic programming
  → e.g., predicate, f(X), p(Y,f(a)) are Prolog atoms
  → e.g., predicate, f(a), p(x,y) are Prolog ground atoms

Towards meta-programmings

- `parent(lino,joey)` – out of context – could represent either a ground atom or a ground term
- Is this an issue or a feature?
- It paves the way towards *meta-programming*
**Prolog Syntax III**

**Prolog clauses**

A **clause** is a Horn clause of the form $A :\hat{} B_1, \ldots, B_n$.

- where $A$, $B_1$, $\ldots$, $B_n$ are Prolog atoms
- $A$ is the **head** of the clause
- $B_1, \ldots, B_n$ is the **body** of the clause
- $:\hat{}$ denotes logic implication
- $.$ is the terminator

A **fact** is a clause with no body $A$. ($n = 0$)

A **rule** is a clause with at least one atom in the body

$$A :\hat{} B_1, \ldots, B_n.$$ ($n > 0$)

A **goal** is a clause with no head and at least one atom in the body

$$:- B_1, \ldots, B_n.$$ ($n > 0$)

- often written as $?- B_1, \ldots, B_n.$
Prolog

Prolog Syntax IV

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
Aim of a Prolog computation

- given a Prolog program \( P \) and the goal \(?- p(t_1,t_2,\ldots,t_m)\) (also called query)
- if \( X_1,X_2,\ldots,X_n \) are the variables in terms \( t_1,t_2,\ldots,t_m \)
- the meaning of the goal is to query \( P \) and find whether there are some values for \( X_1,X_2,\ldots,X_n \) that make \( p(t_1,t_2,\ldots,t_m) \) true

\[ \rightarrow \text{thus, the aim of the Prolog computation is to find a substitution } \sigma = X_1/s_1,X_2/s_2,\ldots,X_n/s_n \text{ such that } P \models p(t_1,t_2,\ldots,t_m)\sigma \]
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
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*
Next in Line...
Types in LP

No types in LP

- in pure LP 3 and + are just symbols, with no specific meaning attached
- and, in pure Prolog as well
- how do we deal, e.g., with natural numbers?
  - Giuseppe Peano shows us the way
Peano axioms, also known as Peano’s postulates, in number theory, five axioms introduced in 1889 by Italian mathematician Giuseppe Peano. Like the axioms for geometry devised by Greek mathematician Euclid (c. 300 bce), the Peano axioms were meant to provide a rigorous foundation for the natural numbers (0, 1, 2, 3, …) used in arithmetic, number theory, and set theory. In particular, the Peano axioms enable an infinite set to be generated by a finite set of symbols and rules.
The five Peano axioms are

1. zero is a natural number
2. every natural number has a successor in the natural numbers
3. zero is not the successor of any natural number
4. if the successor of two natural numbers is the same, then the two original numbers are the same
5. if a set contains zero and the successor of every number is in the set, then the set contains the natural numbers
How do we represent natural numbers in LP?

- In LP
  - The *Herbrand Universe* represents the elements of the domain of discourse
  - The *Herbrand Base* represents the elementary propositions over the domain of discourse

→ In order to represent natural numbers in LP, we use
  - *Constants* and *functors* for natural numbers
  - *Predicates* for relations such as $n \in \mathbb{N}$
### Natural Numbers in LP II

**Constants & functors**

- our domain of discourse is \( \mathbb{N} \), the set of *natural numbers*
- our Herbrand Universe needs to express the elements of \( \mathbb{N} \)
- from Peano axioms, we understand that
  - *zero* is essential
    - we use \( z \) as the *constant* symbol for *zero*
    - it is not something belonging to the language: it is just our *(pre-)*interpretation of symbol \( z \) for our current purposes
  - the notion of *successor* is essential, too
    - we use \( s/1 \) (\( s \) with arity 1) as the *functor* symbol for *successor*
    - if \( N \) is a natural number, then \( s(N) \) is another natural number—precisely, the *successor* of \( N \)
    - *recursive data structure*: the successor of \( s(N) \) is \( s(s(N)) \), which is another natural number

**e.g.** in our pre-interpretation, \( z \) is zero, \( s(z) \) is one, \( s(s(z)) \) is two, ...
Natural Numbers in LP III

Predicates

- we focus on the simple relation $n \in \mathbb{N}$—that is, $n$ is a natural number
- we use `nat/1` (nat with arity 1) as the *predicate* symbol for such a relation
  - if $N$ is a variable, atom `nat(N)` represents the proposition “$N$ is a natural number”
  - according to our current *interpretation*
- e.g. in our interpretation, atom `nat(s(s(s(z))))` means that ‘two is a natural number’
Natural Numbers in Prolog I

A first program

```
nat(z). % zero is a natural number
nat(s(z)). % one is a natural number
nat(s(s(z))). % two is a natural number
...
```

- remarks
  - infinite number of clauses (facts) for infinite numbers
  - not exactly in the spirit of axiomatic systems
  - in particular, of Peano's one
  - we just used facts
  - rules are what triggers intensional representation, with universal variables
  - finite representation for (potentially) infinite propositions
### Natural Numbers in Prolog II

#### A second program: recursive rules

```
nat(z). % zero is a natural number
nat(s(N)) :- nat(N). % if N is a natural number
% its successor is a natural number
```

- **remarks**
  - one fact and one rule for infinite numbers
  - a *recursive* rule

- **goals**
  1. `?- nat(z).`
  2. `?- nat(0).`
  3. `?- nat(s(s(s(s(s(s(z))))))).`
  4. `?- nat(N).`
  5. `?- nat(s(N)).`

---

Andrea Omicini (DISI, Univ. Bologna)
Natural Numbers in Prolog III

- remarks
  - \( z \) is zero, 0 is not zero
  - recursive data structures are powerful and expressive, yet unreadable for humans: which is why Prolog is actually \textit{impure} LP
  - LP allows for \textit{generation} of data—-e.g., the set of natural numbers, or the set of positive numbers
  - \textit{unlimited} number of SLD \textit{refutations}: infinite branch of the proof tree
  - \textit{order of clauses} matters in Prolog: what if we exchange the fact with the rule?
Computing with Naturals in Prolog I

### Sum

- `plus(z, N, N). % zero plus N is N`
- `plus(s(M), N, s(P)) % if M plus N is P, then
  :- plus(M, N, P). % adding N to the successor of M
  % returns the successor of P`

**Remarks**

- based on the same representation of \( \mathbb{N} \) used above
- `plus/3` actually represents an equation: \( \text{plus}(X, Y, Z) \) means that \( X + Y = Z \)
Computing with Naturals in Prolog II

- goals

1. `- plus(s(z), s(s(z)), S).`
2. `- plus(0, s(0), S).`
3. `- plus(X, s(s(z)), s(s(s(s(s(s(z))))))).`
4. `- plus(s(s(s(s(z)))), Y, s(s(s(s(s(s(z))))))).`
5. `- plus(X, Y, s(s(s(s(s(z)))))).`
6. `- plus(X, s(s(z)), Y), plus(X, Y, s(s(s(s(s(s(z))))))).`
Computing with Naturals in Prolog III

- remarks
  - again, \( z \) is zero, 0 is not zero
  - no predefined input/output parameters for \texttt{plus/3} procedure
  - \( \rightarrow \) atoms represent equations
  - generation of possible solutions
  - equations can be combined in systems of equations
Lists

- Lists are defined via two constructors
  - `nil` the empty list, containing no elements
  - `cons` the constructor `cons`, taking an element $H$ and a list $T$, and generating the list `cons(H, T)`

E.g. `cons(a, cons(b, cons(c, nil)))` would represent list $a, b, c$

- Typical recursive data structures
- Used to represent sequences of any sort
Lists in Prolog II

Prolog lists

- in Prolog, list are defined via two analogous constructors
  - [] represents the empty list, containing no elements—a constant
    - stands for cons, taking an element H and a list T, and generating the list .(H,T)—a functor of arity 2
  - Prolog sequence notation simplifies writing lists
    - .(H,T) can be written as [H|T]
    - .(H,.(H’,T’)) can be written as [H,H’|T’]
    - there, empty list can be omitted

e.g. [a,b,c] would represent list a, b, c in Prolog, where
  - a is the head of the list
  - [b,c] is the tail of the list
  - mgu([a,b,c],[H|T]) = {H/a, T/[b, c]}
Computing with Lists I

Recursion

- being recursive data structures, lists are typically handled by recursive rules
- which incidentally is also the only way to handle repeated operations over sequences in Prolog, where there is nothing like a cycle programming construct

Recursion scheme in Prolog

- since Prolog search strategy is depth-first
- in particular, with clauses used orderly, top-down
- termination is handled with a fact, typically coming before the recursive rule
- as already seen in the cases of num/1 and plus/3 above
Typical Example: member/2

Checking whether the first argument is a term that is a member of the list in the second argument

\[
\text{member}(X, [X|Xs]). \\
\text{member}(X, [Y|Ys]) :- \text{member}(X, Ys).
\]

- goals

1. \?- \text{member}(b, [a,b,c])
2. \?- \text{member}(X, [a,b,c]).
3. \?- \text{member}(g(X), [f(a), g(b), f(c), g(d)]).
4. \?- \text{member}(z, [XT]).
Typical Example: member II

- remarks
  - search strategy: left to right through the list
  - devising out all the members of the list
  - conditional membership—given a certain computed substitution
  - generation of lists
What We do not Do

Many interesting things are left out... that are relevant for Prolog programming
- operator definition
- conditionals
- control: cut & fail
- negation as failure (NAF)
- closed world assumption (CWA)
- arithmetic
- meta programming

and many more

however, this is not a Prolog course

and we already discussed whatever could be useful for our purposes


Logic Programming in Prolog with tuProlog

Autonomous Systems / Technologies
Sistemi Autonomi / Tecnologie

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2016/2017