CSC324 — Logic & Relational Programming Illustrated in Prolog

Afsaneh Fazly[1]

Winter 2013

[1]with many thanks to Anya Tafliovich, Gerald Penn and Sheila McIlraith.

Logic Programming and Prolog

Logic PLs are neither procedural nor functional.

- Specify **relations** between objects
  - `larger(3,2)`
  - `father(tom,jane)`

- Separate logic from control:
  - Programmer declares **what** facts and relations are true.
  - System determines **how** to use facts to solve problems.
  - System **instantiates** variables in order to make relations true.

- Computation engine: theorem-proving and recursion (Unification, Resolution, Backward Chaining, Backtracking)
  - Higher-level than imperative (and functional) languages.

Prolog

Suppose we state these **facts**:

```
male(charlie).
female(alice).
male(bob).
female(eve).
```

```
parent(charlie,bob).
parent(eve,bob).
parent(charlie,alice).
parent(eve,alice).
```

We can then make **queries**:

```
?- male(charlie).
true

?- male(eve).
false

?- female(Person).
Person = alice;
Person = eve;
false
```

```
?- parent(Person, bob).
Person = charlie;
Person = eve;
false

?- parent(Person, bob),
   female(Person).
Person = eve;
false
```

Prolog

We can also state **rules**, such as this one:

```
sibling(X, Y) :- parent(P, X),
                 parent(P, Y).
```

Then the queries become more interesting:

```
?- sibling(charlie, eve).
false

?- sibling(bob, Sib).
Sib = bob;
Sib = alice;
Sib = bob;
Sib = alice;
false
```

Logic vs Functional Programming

In FP, we program with **functions**.

- $f(x, y, z) = x * y + z - 2$
- Given a function, we can only ask one kind of question:

  Here are the argument values; tell me what the function's value is.

In LP, we program with **relations**.

- $r(x, y, z) = \{(1, 2, 3), (9, 5, 13), (0, 0, 4)\}$
- Given a predicate, we can ask many kinds of question:

  Here are some of the argument values; tell me what the others have to be in order to make a true statement.

Logic Programming

- A program consists of facts and rules.
- Running a program means asking queries.
- The language tries to find one or more ways to prove that the query is true.
- This may have the side effect of freezing variable values.
- The language determines how to do all of this, *not* the program.
- How does the language do it? Using unification, resolution, and backtracking.

Prolog Syntax

constants and **variables**:

- Constants are:
  - Atoms: any string consisting of alphanumeric characters and underscore (_) that begins with a lowercase letter.
    E.g.: charilie, bob, a_book, book_n10, ...
  - Numbers.
    E.g.: 1, -2, 6.02e-23, 99.1, ...
- Variables are strings that begin with '_' or an uppercase letter.
  E.g., Person, _X, _, A_very_long_var_name, ...

Example: in

```
parent(Person, bob).
```

`bob` is a Constant, and `Person` is a Variable.

Prolog Syntax

```
<const>  ::= <atom> | <number>
<var>    ::= <ucase> <string> | _<string>
<atom>   ::= <lcase> <string>
<letter> ::= <ucase> | <lcase>
<string> ::= epsilon | <letter><string> |
             <number><string> | _<string>
<ucase>  ::= A | ... | Z
<lcase>  ::= a | ... | z
<number> ::= ...
```

Prolog Syntax

terms are inductively defined:

- Constants and variables are terms.
- Compound terms – applications of any n-ary **functor** to any n terms – are terms.

```
author(jane_austen)
book(persuation,author(jane_austen))
```

- Nothing else is a term.

terms denote entities in the world.

Note: Prolog distinguishes numbers and variables from other terms in certain contexts, but is otherwise untyped/monotyped.

Prolog Syntax & Semantics

```
<term> ::= <const> | <var> |
           <functor> ‘(’ <term> { , <term> } ‘)’
```

terms denote entities in the ‘world’.

Prolog Syntax

atomic formulae consist of an *n*-ary relation (also called **predicate**) applied to *n* terms (also called **arguments**).

Note: syntactically, formulae look like terms because of Prolog's weak typing. But formulae are either true or false; terms denote entities in the world.

In Prolog, atomic formulae can be used in three ways:

- As **facts**: in which they assert truth,
  e.g., `own(mary,book(persuation,author(jane_austen)))`
- As **queries**: in which they enquire about truth,
  e.g., `own(mary,book(_,author(_)))`
- As components of more complicated statements (see below).

Prolog Syntax

```
<term> ::= <const> | <var> |
           <functor> ‘(’ <term> { , <term> } ‘)’

<pred> ::= <pname> ‘(’ <term> { , <term> } ‘)’
```

terms denote entities in the ‘world’.

predicates speficy relations among entities.

Prolog Queries

A **query** is a proposed fact that is to be proven.

- If the query has no variables, returns true/false.
- If the query has variables, returns appropriate values of variables (called a substitution).

```
?- male(charlie).
true

?- male(eve).
false

?- female(Person).
Person = alice;
Person = eve;
false
```

Prolog Syntax

complex formulae are formed by combining simpler formulae. We only discuss conjunction and implication.

- conjunction: in Prolog, `and` looks like `','`

```
?- parent(Person, bob), female(Person).
Person = eve;
false
```

conjunctions can be used as queries, but not as facts. Using multiple facts is an implicit conjunction.

Prolog Syntax

- implication: in Prolog, these are very special.

They are written backwards (conclusion first), and the conclusion must be an atomic formula. This backwards implication is written as '`:-`', and is called a **rule**.

```
sibling(X, Y) :- parent(P, X), parent(P, Y).
```

Rules can be used in programs to assert implications, but not in queries.

Prolog and Horn Clauses

Recall:

- Prolog program: a set of **facts** and **rules**.
- Running a program: asking **queries**.
- System/Language tries to prove that the query is true.

Note that the Prolog system neither understands the facts and rules, nor can think or reason about them.

Prolog system builds proofs by making inferences based on a resolution theorem-prover for **Horn clauses** in first-order logic.

Horn Clauses

A Horn clause is: $c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \ldots \wedge h_n$

- Consequent c: an atomic formula.
- Antecedents: zero or more atomic formulae (h_i) conjoined.

Meaning of a Horn clause:

- "The consequent is true if the antecedents are all true"
- c is true if h_1, h_2, h_3, ..., and h_n are all true

Horn Clause

In Prolog, a Horn clause

$$c \leftarrow h_1 \wedge \ldots \wedge h_n$$

is written

```
c :- h1, ..., hn.
```

- Horn clause = Clause
- Consequent = Goal = Head
- Antecedents = Subgoals = Tail
- Horn clause with No tail = Fact
- Horn clause with tail = Rule
- Horn clause with No head = Query

Prolog Horn Clause

A Horn clause with no tail:

```
male(charlie).
```

i.e., a **fact**: `charlie` is a male dependent on no other conditions.

A Horn clause with a tail:

```
father(charlie,bob) :-
        male(charlie), parent(charlie,bob).
```

i.e., a **rule**: `charlie` is the father of `bob` if `charlie` is male and `charlie` is a parent of `bob`'s.

Prolog Horn Clause

A Horn clause with no head:

```
?- male(charlie).
```

i.e., a **query**: is it true that `charlie` is a male?

Note that this corresponds to the head-less Horn clause:

```
:- male(charlie).
```

Syntax of a Prolog Program

A logic program (a theory that needs to be proven) is a collection of Horn clauses.

A simplified Prolog grammar:

```
<clause> ::= <pred> . |                              <-- fact
             <pred> :- <pred> { , <pred> } .         <-- rule

<pred>   ::= <pname>‘(’ <term> { , <term> } ‘)’

<term>   ::= <functor>‘(’ <term> { , <term> } ‘)’
             | <const> | <var>

<const>  ::= ...
<var>    ::= ...
```

Meaning of Prolog Rules

A prolog rule must have the form:

```
c :- a1, a2, a3, ..., an.
```

which means in logic:

$$a_1 \wedge a_2 \wedge a_3 \wedge \cdots \wedge a_n \rightarrow c$$

Restrictions

- There can be zero or more antecedents, but they are conjoined; we cannot disjoin them.
- There cannot be more than one consequent.

non-Horn clauses

Many non-Horn formulae can be converted into logically equivalent Horn-formulae, using propositional tautologies, e.g.:

$\neg\neg a \Leftrightarrow a$	double negation
$\neg(a \vee b) \Leftrightarrow \neg a \wedge \neg b$	DeMorgan
$a \vee (b \wedge c) \Leftrightarrow (a \vee b) \wedge (a \vee c)$	distributivity
$a \rightarrow b \Leftrightarrow \neg a \vee b$	implication

Bending the Restrictions?

Getting disjoined antecedents

Example: $a_1 \vee a_2 \vee a_3 \rightarrow c$.

Solution?

Syntactic sugar: ;

Getting more than 1 consequent, conjoined

Example: $a_1 \wedge a_2 \wedge a_3 \rightarrow c_1 \wedge c_2$.

Solution?

Getting more than 1 consequent, disjoined

Example: $a_1 \wedge a_2 \wedge a_3 \rightarrow c_1 \vee c_2$.

Solution?

Variables

Variables may appear in antecedents & consequent of Horn clauses.

Prolog interprets free variables of a rule **universally**.

$$\texttt{c(X}_1\texttt{,...,X}_n\texttt{) :- f(X}_1\texttt{,...,X}_n\texttt{,Y}_1\texttt{,...,Y}_m\texttt{)}$$

is, in first-order logic:

$$\forall X_1, \ldots, X_n, Y_1, \ldots, Y_m \cdot c(X_1, \ldots, X_n) \leftarrow f(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$$

or, equivalently:

$$\forall X_1, \ldots, X_n \cdot c(X_1, \ldots, X_n) \leftarrow \exists Y_1, \ldots, Y_m \cdot f(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$$

Variables

Because of this equivalence, we sometimes say that free variables that do not appear in the head are quantified **existentially**.

Recall that a Prolog query corresponds to a head-less Horn clause.

Thus, the Prolog **query** `?-q(X1, ..., Xn)` means

$$\exists X_1, \ldots, X_n \cdot q(X_1, \ldots, X_n)$$

That is, the Prolog system tries to find whether there exist values for the variables $X_1 \ldots X_n$ such that $q(X_1, \ldots, X_n)$ is true.

Horn Clauses with Variables

```
isaMother(X) :- female(X), parent(X, Y).
```

In first-order logic:

$$\forall X \cdot \mathit{isaMother}(X) \leftarrow \exists Y \cdot \mathit{parent}(X, Y) \wedge \mathit{female}(X).$$

swi-prolog interface on CDF

```
% swipl
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.4)
...
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- ['family'].                          <--- load file family.pl
% family compiled 0.00 sec, 2,856 bytes
true.

?- parent(Person, bob).
Person = charlie ;                      <--- to get more answers
Person = eve ;
false.

?-                                      <--- query mode
                                           can only ask quires
                                           no facts/rules can be added
```

swi-prolog interface on CDF

```
?- [user].                                    <--- enter predicate mode

|: grandmother(X,Z) :- grandparent(X,Z), female(X).
|: grandfather(X,Z) :- grandparent(X,Z), male(X).
|: % user://1 compiled 0.00 sec, 652 bytes   <--- to exit & compile,
true.                                              press Ctrl+D

?- grandmother(_,alice).
true.

?- grandmother(X,alice).
X = mary.

?- grandparent(Z,alice).
Z = frank ;
Z = mary.

?-
```

Horn Clauses with Variables

```
% swipl
Welcome to SWI-Prolog ...

?- ['family2'].
Warning: /u/afsaneh/prolog/family2.pl:24:
        Singleton variables: [Y]
% family2 compiled 0.00 sec, 3,032 bytes
true.

?- isaMother(X).
X = eve;
X = eve;
false
```

No singleton variables:

```
isaMother(X) :- female(X), parent(X, _).
```

Execution of Prolog Programs

- **Unification**: variable bindings.

- **Backward Chaining/**
  Top-Down Reasoning/
  Goal-Directed Reasoning:
  Reduces a goal to one or more subgoals.

- **Backtracking**:
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.

Unification

Two atomic formulae unify if and only if they can be made syntactically identical by replacing their variables by other terms.

- `parent(bob,Y)` unifies with `parent(bob,sue)` by replacing `Y` by `sue`.

```
?- parent(bob,Y)=parent(bob,sue).
Y = sue.
```

- `parent(bob,Y)` unifies with `parent(X,sue)` by replacing `Y` by `sue` and `X` by `bob`.

```
?- parent(bob,Y)=parent(X,sue).
Y = sue
X = bob.
```

Unification

- A **substitution** is a function that maps variables to Prolog terms, e.g., `{ X\sue, Y\bob }`

- An **instantiation** is an application of a substitution to all of the free variables in a formula or term.

  If `S` is a substitution and `T` is a formula or term, then `ST` (or `S(T)`) denotes the instantiation of `T` by `S`.

```
T = parent(X,Y)
S = { X\sue, Y\bob }
ST = parent(sue,bob)

T = likes(X,X)
S = { X\bob }
ST = likes(bob,bob)
```

Unification

- C is a **common instance** of formulae/terms A and B, if there exist substitutions S1 and S2 such that C = S1(A) = S2(B).

```
A = parent(bob,X), B = parent(Y,sue)
S1 =  {  X\sue }, S2 = { Y\bob }
S1(A) = S2(B) = parent(bob,sue) = C
```

Unification

- C is a **common instance** of formulae/terms A and B, if there exist substitutions S1 and S2 such that C = S1(A) = S2(B).

```
A = parent(bob,X), B = parent(Y,sue)
S1 =  {  X\sue }, S2 = { Y\bob }
S1(A) = S2(B) = parent(bob,sue) = C
```

- A and B are **unifiable** if they have a common instance C. A substitution that produces a common instance is called a **unifier** of A and B.

```
parent(bob,X)  and  parent(Y,sue) are unifiable:
{  X\sue, Y\bob } is the unifier
```

Unification

Examples:

`p(a,a) ? p(a,a)`

`p(a,b) ? p(a,a)`

`p(X,X) ? p(b,b)`
`p(X,X) ? p(c,c)`
`p(X,X) ? p(b,c)`

`p(X,b) ? p(Y,Y)`

`p(X,Z,Z) ? p(Y,Y,b)`

`p(X,b,X) ? p(Y,Y,c).`

Unification

Examples:

`p(a,a)` unifies with `p(a,a)`.

`p(a,b)` does not unify with `p(a,a)`.

`p(X,X)` unifies with `p(b,b)` and with `p(c,c)`, but not with `p(b,c)`.

`p(X,b)` unifies with `p(Y,Y)` with unifier `{ X\b, Y\b }` to become `p(b,b)`.

`p(X,Z,Z)` unifies with `p(Y,Y,b)` with unifier `{ X\b, Y\b, Z\b }` to become `p(b,b,b)`.

`p(X,b,X)` does not unify with `p(Y,Y,c)`.

Unification

Examples:

- `p(f(X),X)` unifies with `p(Y,b)`
  with unifier $\{$`X\b, Y\f(b)`$\}$
  to become `p(f(b),b)`.

- `p(b,f(X,Y),c)` unifies with `p(U,f(U,V),V)`
  with unifier $\{$`X\b, Y\c, U\b, V\c`$\}$
  to become `p(b,f(b,c),c)`.

Unification

`p(b,f(X,X),c)` does *not* unify with `p(U,f(U,V),V)`.
Reason:

- To make the first arguments equal,
  we *must* replace `U` by `b`.
- To make the third arguments equal,
  we *must* replace `V` by `c`.
- These substitutions convert
  `p(U,f(U,V),V)` into `p(b,f(b,c),c)`.
- However, *no* substitution for `X` will convert
  `p(b,f(X,X),c)` into `p(b,f(b,c),c)`.

Unification

`p(f(X),X)` should *not* unify with `p(Y,Y)`.
Reason:

- Any unification would require that
  `f(X)` = `Y` and `Y` = `X`
- But no substitution can make
  `f(X)` = `X`

However, Prolog claims they unify:

```
?- p(f(X),X) = p(Y,Y).
X = Y, Y = f(Y).

?-
```

Unification

If a rule has a variable that appears only once, that variable is called a "singleton variable".

Its value doesn't matter — it doesn't have to match anything elsewhere in the rule.

```
isaMother(X) :- female(X), parent(X, Y).
```

Such a variable consumes resources at run time.

We can replace it with `_`, the **anonymous variable**. It matches anything. If we don't, Prolog will warn us.

Note that `p(_,_)` unifies with `p(b,c)`. Every instance of the anonymous variable refers to a different, **unique** variable.

Most General Unifier (MGU)

The atomic formulas `p(X,f(Y))` and `p(g(U),V)` have infinitely many unifiers.

- {`X\g(a), Y\b, U\a, V\f(b)`}
  unifies them to give `p(g(a),f(b))`.
- {`X\g(c), Y\d, U\c, V\f(d)`}
  unifies them to give `p(g(c),f(d))`.

Most General Unifier (MGU)

The atomic formulas `p(X,f(Y))` and `p(g(U),V)` have infinitely many unifiers.

- {`X\g(a), Y\b, U\a, V\f(b)`}
  unifies them to give `p(g(a),f(b))`.
- {`X\g(c), Y\d, U\c, V\f(d)`}
  unifies them to give `p(g(c),f(d))`.

However, these unifiers are more specific than necessary.

Their **most general unifier** (MGU) is {`X\g(U), V\f(Y)`} that unifies the two atomic formulas to give `p(g(U),f(Y))`.

Every other unifier results in an atomic formula of this form.

The MGU uses variables to fill in as few details as possible.

Most General Unifier

Example:

$$f(W,\ g(Z),\ Z)$$

$$f(X,\ Y,\ h(X))$$

To unify these two formulae, we need

$$\begin{array}{lcl} Y & = & g(Z) \\ Z & = & h(X) \\ X & = & W \end{array}$$

Working backwards from W, we get

$$\begin{array}{lcl} Y & = & g(Z) = g(h(W) \\ Z & = & h(X) = h(W) \\ X & = & W \end{array}$$

So, the MGU is

$$\{X \backslash W,\ Y \backslash g(h(W)),\ Z \backslash h(W)\}$$

Most General Unifier

The substitution that results in the most general instance is called the **most general unifier** (MGU).

It is **unique**, up to consistent renaming of variables.

MGU is the one that Prolog computes.

A substitution σ is the MGU of a set of expressions E if:

- it unifies E,
  and
- for any unifier ω of E, there is a unifier λ, such that $\omega(E) = \lambda \circ \sigma(E)$.

Most General Unifier

E.g., given `p(X,f(Y))` and `p(g(U),V)`,
an MGU σ = `{ X\g(U), V\f(Y) }`,
and a unifier ω=`{ X\g(a), Y\b, U\a, V\f(b) }`,

we have ω`(p(X,f(Y))) = p(g(a),f(b))`,

σ`(p(X,f(Y))) = p(g(U),f(Y))`,
so exists λ=`{ U\a, Y\b }`,
so that ω`(p(X,f(Y)))` $= \lambda(\sigma$`(p(X,f(Y)))`$)$

also ω`(p(g(U),V)) = p(g(a),f(b))`,

σ`(p(g(U),V)) = p(g(U),f(Y))`,
so for λ=`{ U\a, Y\b }`,
we have ω`(p(g(U),V))` $= \lambda(\sigma$`(p(g(U),V))`$)$

Most General Unifier

Examples:

t_1	t_2	MGU
f(X,a)	f(a,Y)	
f(h(X,a),b)	f(h(g(a,b),Y),b)	
g(a,W,h(X))	g(Y,f(Y,Z),Z)	
f(X,g(X),Z)	f(Z,Y,h(Y))	
f(X,h(b,X))	f(g(P,a),h(b,g(Q,Q)))	