Logic Programming

November 23, 2019

Introduction to Logic Programming

Truly Declarative Paradigm

- User declares **what facts** are true.
- User states some queries.
- System determinines **how to use the facts** that are true to answer the queries.

Primary difference between imperative programming and logic programming.

- Imperative: explicitly instruct the system how certain computation should be performed.
- Logic: instruct the system what can be used to perform some computation.

demo some examples

Prolog

Facts

```
isamother(mary).          %% Horn Clause with no Antecedent
childof(tom, mary).       %% Horn Clause with no Antecedent
```

Rules

```
%% Rule: Horn Clause with antecedent
loves(mary, tom) :-
        isamother(mary), childof(tom, mary).
```

Query

```
%% Query: Horn Clause with no consequent
?- loves(mary, tom).
```

?- loves(X, tom).
mary
?- loves(mary, Y).
tom
?- loves(mary, jane).
false

What is Logic Programming

Overview: there are many (overlapping) perspectives on logic programming:

- A very high level programming language
- An interpretation of *declarative specifications*
- Non-procedural programming
- Algorithms minus control
- Computations as *deduction*
- Theorem proving

A Very High Level Language

- A good programming language should not encumber the programmer with non-essential details.
- The development of programming languages has been toward freeing the programmer of more and more of the details
  - ASSEMBLY LANGUAGE: symbolic encoding of data and instructions.
  - FORTRAN: allocation of variables to memory locations, register saving, etc.
  - ML: explicit variable type declarations
  - JAVA: Platform specifics
- Logic Programming Languages are a class of languages which attempt to free us from having to worry about many aspects of explicit control.

An Interpretation of Declarative Specifications

- Logical statement: Forall X and Y, X is the father of Y if X is a parent of Y and the gender of X is male.
- Prolog code:
```
father(X,Y) :-
        parent(X,Y), gender(X,male).
```
- Interpret it in two slightly different ways:
  - declaratively - which must be true if a father relationship holds.
  - procedurally: what to do to establish that a father relationship holds.

Non-procedural Programming

- A non - procedural language one in which one specifies WHAT needs to be computed but not HOW it is to be done.
- it specifies a state with constraints, objects and relations:
  - the set of objects involved in the computation
  - the relationships which hold between them
  - the constraints which must hold for the problem to be solved
- the language interpreter or compiler will decide HOW to satisfy the constraints.

Algorithms Minus Control

- Nikolas Wirth (architect of Pascal) used the following slogan as the title of a book: Algorithms + Data Structures = Programs
- Bob Kowalski offers a similar one to express the central theme of logic programming: Algorithms = Logic + Control
- We can view the LOGIC component as: A specification of the essential logical constraints of a particular problem
- CONTROL component as: Advice to an evaluation machine (e.g. an interpreter or compiler) on how to go about satisfying the constraints)

Computation as Deduction

- Computation is related to logical proofs and is not restricted to functional (Church) or imperative (Turing/Von Neumann) computation models.
  - inductive reasoning: particular cases to general cases - inductive reasonable and proof
  - deductive reasoning:
    All men are mortal. (First premise)
    Socrates is a man. (Second premise)
    Therefore, Socrates is mortal. (Conclusion)
- It uses the language of logic to express data and programs, e.g., Forall X and Y, X is the father of Y if X is a parent of Y and the gender of X is male.
- Current logic programming languages use first order logic (FOL)
- Propositions, e.g., A is father of B, predicates, e.g., parent (X Y), and quantifier symbols such as $\exists$ and $\forall$ on objects (more in discrete maths books).

Theorem Proving

- Logic programming uses the notion of an automatic theorem prover as an interpreter.
- The theorem prover derives a desired solution from an initial set of axioms.
- Note that the proof must be a "constructive" one so that more than a true/false answer can be obtained.
- E.G. The answer to exists x such that x = sqrt(16) should be x = 4 or x = -4 rather than true

A Short History

1965 Efficient theorem provers. Resolution (Alan Robinson)
1969 Theorem Proving for problem solving. (Cordell Green)
1969 PLANNER, theorem proving as programming (Carl Hewett)
1970 Micro - Planner, an implementation (Sussman, Charniak and Winograd)
1970 Prolog, an implementation (Alain Colmerauer)
1972 Book: Logic for Problem Solving. (Kowalski)
1977 DEC - 10 Prolog, an efficient interpreter/compiler (Warren and Pereira)
1982 Japan's 5th Generation Computer Project
1985 Datalog and deductive databases
1995 Prolog interpreter embedded in NT

PROLOG is the FORTRAN of Logic Programming

- Prolog is the only widely used logic programming language.
- As a Logic Programming language, it has a number of advantages: simple, small, fast, easy to write good compilers for it.
- and disadvantages
  - It has a fixed control strategy.
  - It has a strong procedural aspect
  - limited support parallelism or concurrency or multi-threading.

Interpreter of Prolog

- "Execute a program": make inference from a database of facts and rules.
- Presenting knowledge: predicate logic
  - fact: proposition that's unconditional true
  - rule: proposition that's conditional true; dependent on other propositions
  - a fact or a rule is a statement or clause in Prolog.

Some Underlying Ideas

To Understand Computing with Logic

We need to understand Logic.

Logic

- Declarative Statements describing the state of the world.
  - Declarative Statements are either true or false
- Rules of Reasoning use existing declarative statements to conclude new declarative statements.

Basic Constituents of Logic

1. Individuals in the world (constants).
2. Relations over these individuals (properties or predicates): E.g., Edges between nodes.
   - Relations have arity (number of individuals involved in the relation)
     Facts and Rules.
     E.g., n is a node, n has an edge to n'.
3. Quantifiers and variables used to describe all or some individuals

Some Underlying Ideas

Another Example

Declarative Statements: Facts and Rules

1. Every mother loves her children.
2. Mary is a mother and Tom is Mary's child.

Queries

Does Mary love Tom?

Some Underlying Ideas

Another Example

- Constants: *mary*, *tom*, ...
- Predicates: isamother/1, childof/2, loves/2

Declarative Statements: Facts and Rules

Facts Mary is a mother isamother(*mary*)

Facts Tom is Mary's child. childof(*tom*, *mary*)

Rule Every mother loves her children.

$\forall X.\forall Y.(\mathrm{loves}(X, Y) \quad (\mathrm{isamother}(X) \wedge \mathrm{childof}(Y, X)))$

Queries:

Does Mary love Tom?

$\mathit{true} \overset{?}{\quad} \mathrm{loves}(\mathit{mary}, \mathit{tom})$ true

Some Underlying Ideas

Horn Clause

Alfred Horn

A Horn Clause:

$$c \qquad h_1 \wedge h_2 \wedge \ldots \wedge h_n$$

where c is the consequent and the conjunction of h_is is the antecedent.

Some Underlying Ideas

Horn Clause

Alfred Horn

A Horn Clause:

$$c \qquad h_1 \wedge h_2 \wedge \ldots \wedge h_n$$

where c is the consequent and the conjunction of h_is is the antecedent.

If all h_is are true then c is true

Some Underlying Ideas

Horn Clause: $c \quad \bigwedge_i h_i$

A Horn clause

$$c \leftarrow h_1 \wedge h_2 \wedge \ldots h_n$$

is written in prolog as

```
c :- h1, h2, ..., hn
```

Horn Clause	Prolog
Consequent c	goal
Antecedent $\bigwedge_i h_i$	subgoals
Horn Clause with no Antecedent	Fact
Horn Clause with Antecendent	Rule
Horn Clause with no Consequent	Query

Some Underlying Ideas

Prolog

Facts

```
isamother(mary).         %% Horn Clause with no Antecedent
childof(tom, mary).      %% Horn Clause with no Antecedent
childof(jerry, mary).    %% Horn Clause with no Antecedent
```

Rules

```
%% Rule: Horn Clause with antecedent and with variables
%%       X and Y are universally quantified
loves(X, Y) :-
        isamother(X), childof(Y, X).

%%       X is universally quantified
%%       Y, Z are existentially quantified
hassibling(X) :-
        childof(X, Y), childof(Z, Y).
```

Query

```
%% Query: Horn Clause with no consequent
?- loves(mary, X). %% X is exitentially quantified
?- hassibling(jerry).
```

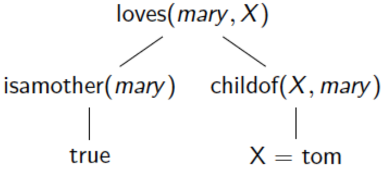
Some Underlying Ideas

Queries with variables

Queries:

```
?- loves(mary, X).
```

means: does there exists an X such that loves($mary, X$) is true.



Queries with free variables will generate a binding for free variables

Some Underlying Ideas

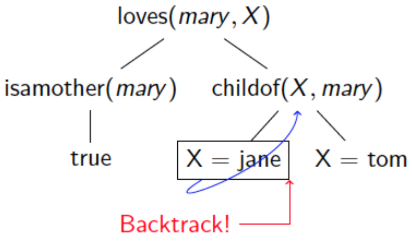
Queries with variables

```
isamother(mary).
childof(jane, mary).
childof(tom, mary).

loves(X, Y) :- isamother(X), childof(Y, X).
```

What is the result of `?- loves(mary, X)`

Computes all possible way to satisfy $\exists X.\text{loves}(mary, X)$.



Some Underlying Ideas

Syntax of Logic Program

Logic program is a collection of Horn Clauses

- How do we write

$$c \leftarrow h_1 \vee h_2$$

as a Horn Clause Statement

$$c \leftarrow h_1$$
$$c \leftarrow h_2$$

```
edge(a, b).
edge(b, c).
edge(c, c).

reach(X, Y) :- edge(X, Y).
reach(X, Y) :- edge(X, Z), reach(Z, Y).
```

?-reach(a, X)
b, c

Some Underlying Ideas

Syntax of Logic Programs

Terms

constants, Variables, functors (un-interpreted functions with terms as arguments)

Formulas

predicates with terms as arguments, boolean combination of predicates and universal/existentially quantified variables followed by predicates.

Some Underlying Ideas

Syntax of Logic Programs

```
Clause -> Predicate.                        <-- fact
        | Predicate :- PredicateSeq.        <-- rule
PredicateSeq -> Predicate
              | Predicate, PredicateSeq

Predicate -> PredName(TermSeq)
TermSeq    -> Term
            | Term, TermSeq

Term       -> FunctorName(TermSeq) | Constant | Variable
```

Some Underlying Ideas

Computing with Logic

Example

```
odd(succ(0)).   %% fact using a Functor
odd(succ(succ(Z))) :- odd(Z). %% rule using a Functor

?- odd(X).   %% Query
```

odd(X)

$X = succ(0)$ $\quad$ $X = succ(succ(Z)) \wedge odd(Z)$

$X = succ(succ(Z)) \wedge Z = succ(0)$ $\quad$ $X = succ(succ(Z)) \wedge Z = succ(succ(Z1)) \wedge odd(Z1)$

...

```
?- odd(X), odd(succ(X)). %% Query: what is the result?
```

Result: succ(0)

Some Underlying Ideas

Logical Implications

- Prolog input: facts and rules (relations)
- Queries: does some inference hold?
- Proof by application logical implication

Resolution

Horn clauses: a rule-based logical formula

Some Underlying Ideas

Executing Logic Programs

Unification/Most General Unifiers

Variable bindings

Backward Chaining/Goal-directed Reasoning

Reducing one proof obligation (goal) into simpler ones (subgoals).

Backtracking

Search for proofs (answers).

Some Underlying Ideas

Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?
  yes by replacing `X` by `mary`
- Unify `childof(jane, X)` and `childof(jane, Y)`?
  yes by replacing `X` and `Y` by the same individual
- Unify `childof(jane, X)` and `childof(Y, mary)`?
  yes by replacing `X` by `mary`, and `Y` by `jane`
- Unify `childof(jane, X)` and `childof(tom, Y)`? No.

Some Underlying Ideas

Substitution

Substitution maps variables to terms.
Instantiation is the application of substitution to all variables in a prolog formula, term.

- Unify `childof(jane, X)` and `childof(Y, mary)`?
  yes by [`X` $\mapsto$ `mary`, `Y` $\mapsto$ `jane`]
- Unify `p(f(X), X)` and `p(Y, a)`?
  yes by [`X` $\mapsto$ `a`, `Y` $\mapsto$ `f(a)`]

Recall, term can be constant, variable, functor.

Some Underlying Ideas

Most General Unifier

MGU results from a substitution that bounds free variables as little as possible

- Unify `p(X, f(Y))` and `p(g(Z), W)`
  - [$X \mapsto g(a)$, $Y \mapsto b$, $W \mapsto f(b)$]
  - [$X \mapsto g(Z)$, $W \mapsto f(Y)$] MGU
- Unify `f(W, g(Z), Z)` and `f(X, Y, h(X))`

MGU?

Soln: [$W \rightarrowtail X$, $Y \rightarrowtail g(Z)$, $Z \rightarrowtail h(x)$] MGU
Or, [$W \rightarrowtail X$, $Y \rightarrowtail g(h(x))$, $Z \rightarrowtail h(x)$] MGU

Some Underlying Ideas

Unification and Computing with Logic

Given a query (prove/disprove a predicate holds)

- Search the facts and rules to find whether the query unifies with any consequent
- If the search fails, return false (query result)
- If the search is successful, then
  - if the unification occurs with the consequent of a fact, return the substitution of the variables (if any)
  - if the unification occurs with the consequent of a rule, instantiate the variables (if any) and prove the subgoals

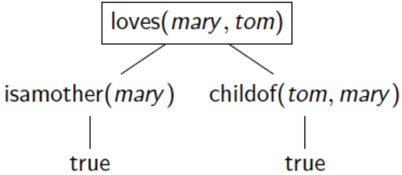
Some Underlying Ideas

Example: Recap

Facts, Rules

```
isamother(mary).
childof(tom, mary).

loves(X, Y) :-
        isamother(X),
         childof(Y, X).
```



Query unifies with the rule
MGU: $[X \mapsto \texttt{mary}, Y \mapsto \texttt{tom}]$.

Queries

```
?- loves(mary, tom).
Yes
```

Some Underlying Ideas

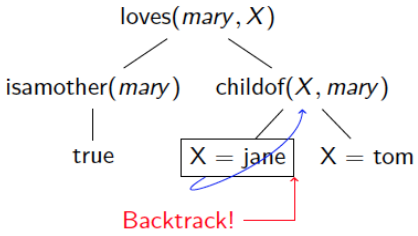
Backtracking: Example Recap

```
isamother(mary).
childof(jane, mary).
childof(tom, mary).

loves(X, Y) :- isamother(X), childof(Y, X).
```

What is the result of `?- loves(mary, X)`

Computes all possible way to satisfy $\exists X.\text{loves}(mary, X)$.



Some Underlying Ideas

Search for solution

```
edge(a, b).
edge(b, c).
edge(c, a).

reach(X, Y) :-
       edge(X, Y).
reach(X, Y) :-
       edge(X, Z),
       reach(Z, Y).

?- reach(a, c)
```

Some Underlying Ideas

More Language Features

- Lists
- Numbers
- if-then-else

Some Underlying Ideas

Lists

A list is ordered sequence of terms enclosed in `[...]`

- `[a, b, c]`: list containing three elements/atoms a, b and c
- `[]`: empty list
- `[a, [b, c], [[d, e]], []]`: list can contain elements of different types
- `[a|[b, c]]`: same as `[a, b, c]`, a is called the head of the list and `[b, c]` is the tail of the list

```
?- [1, 2, 3] = [X|Xs].
X = 1
Xs = [2, 3]

?- [1, 2, 3] = [X|[Y|Rest]].
X = 1
Y = 2
Rest = [3]
```

Some Underlying Ideas

Example

Append one list to another

- appending an empty list L_1 to list L_2 results in L_2
- appending a non-empty list L_1 to list L_2 results in L if the head of L_1 and L are the same and the tail of L is obtained by appending the tail of L_1 to list L_2

If $\mathcal{L}$ represents the set of lists, then signature of append is

$$\mathcal{L} \times \mathcal{L} \times \mathcal{L} \subseteq \mathit{append}$$

```
append([], L, L).

append([X|Xs], L, [X|Ys]) :-
       append(Xs, L, Ys).
```

Some Underlying Ideas

Example

Reverse a list (again!)

```
reverse([], []).
reverse([X|Xs], L) :- reverse(Xs, Ys), append(Ys, [X], L).
```

Length of a list

```
length([], 0).
length([X|Xs], N) :- length(Xs, M), N is M + 1.
```

How about?

```
length([], 0).
length([X|Xs], N) :-  M is N - 1, length(Xs, M).
```

Some Underlying Ideas

Unification vs Computation

- `X = 3`: X is unified to 3 (assignment w/o computation)
- `X = 3 + 1`: X is unified to $3 + 1$ (not 4)
- `X is 3 + 1`: X is assigned to 4

```
?- X is Y + 1.
Uninstantiated argument of evaluable function +/2

?- X is 3, X = 3.
X = 3

?- X = 3, Y is X + 1.
X = 3
Y = 4

?- X is 3, X is X + 1.
no
```

Some Underlying Ideas

if-then-else

```
?- Z = 3, (Z == 3 -> X = 1, Y = 2; X = 2, Y = 1).
Z = 3
X = 1
Y = 2
```

```
 mypred(Z, X, Y) :-
      (Z == 3
       ->   X = 1,
            Y = 2
       ;    X = 2,
            Y = 1
      ).
```

```
mypred(Z, X, Y) :-
     Z == 3, X = 1, Y = 2.
mypred(Z, X, Y) :-
     Z \= 3, X = 2, Y = 1.
```

Example

sentence → noun-phrase verb-phrase .
noun-phrase → article noun
article → a | the
noun → girl | dog
verb-phrase → verb noun-phrase
verb → sees | pets

4. (30 pt) Consider the simple grammar above. Write a Prolog program that parses sentences (represented as lists of words) using the grammar.

This grammar states that a sentence consists of a noun phrase, followed by a verb phrase, followed by a period. It also states that an article is either the word a or the word the.

Hint: A list of words is a sentence if the list is obtained by appending a list which is a noun phrase, a list which is a verb phrase, and a list whose single element is a period. Your program can be used to check if a given sentence, i.e., parse, can be generated by the grammar.

Here is an example interpreter session:

```
| ?- sentence( [ the, girl, sees, a, dog, '.' ] ) .

true ?
```

Example

Sol

```
sentence([]).
sentence([A,B|Tail]):- noun-phrase(A,B),checkVerbPhrase(Tail).
checkVerbPhrase([A,B,C|Tail]):- verb-phrase(A,B,C),checkPeriod(Tail).
checkPeriod([Head|Tail]):- end(Head), isNull(Tail), sentence(Tail).


noun-phrase(A, B):- article(A),noun(B).
verb-phrase(A,B,C) :- verb(A),noun-phrase(B,C).
isEnd(A):-end(A).
isNull([]).


article(a).
article(the).
noun(girl).
noun(dog).
verb(pets).
verb(sees).
end('.').
```

Logic Programming

- numbers: max
- list: append, reverse
- constraint problems
- language parsing
- graph search problems