Notes: Prolog, ILP, ALP, TAL, ASP and examples. Simulators.

Vitalii Protsenko

1 Prolog

1.1 Basics

- Based on rules, ie `dog(a).` means the fact `'a is a dog'`.
- Rules can be derived from other rules by implications `:-` or, more naturally, $\leftarrow$. For example we can write `animal(A) :- dog(A).`, which would mean `A is an animal` $\leftarrow$ `A is a dog`.
- Worth mentioning that A is a variable, meaning that $\forall A : A$ is a dog $\rightarrow A$ is an animal.
- We can write a more complex rules by using $\wedge$, which in Prolog syntax is just a comma. For example: `sister(X, Y) :- parent(Z, X), parent(Z, Y), female(X).` in first-order logic will look like $\forall x \forall y \forall z$: sister(X,Y) $\leftarrow$ parent(Z, X) $\wedge$ parent(Z, Y) $\wedge$ female(X)

1.2 Recursion

We may want to define some more complex family relations, like `grandparent(X,Z) :- parent(X, Y), parent(Y, Z).`, but what about grand-grand parent etc . . . ? Luckily, Prolog allows us to use recursive definitions; let us define `predecessor`:

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

Notice that the base case is necessary if X is a direct parent of Z.
Also, we need to be careful of infinite looping, as the rule `r(A) :- r(A).`, for example, is quite dangerous.

1.3 Variables

As seen before, variables always start with an upper-case letter or an underscore eg `_x` is a valid Prolog variable; furthermore, Prolog renames all the upper-case user defined variables into the underscore-type one automatically for us while executing the code.
We can also use anonymous variable - underscore, and the value of that variable is not output when Prolog answeres the question:

```
?- parent(X, _).
```

will ask whether X is a parent of anybody.

1.4 Structures

We can define some structures like `date(Day, Month, Year).`, as it is easy to make queries on that format, eg:

```
date(13, july, 2013).
?- date(Day, july, 2013).
    Day = 13.
```

1.5 Lists

List is a simple data structure, a sequence of any number of items. In Prolog a typical list can be written like that:

```
[1, 2, dog, sun]
```

and, of course, this is just a syntactical sugar for

```
.(1, .(2, .(dog, .(sun, []))))
```

It is also possible to have nested lists

```
[I, [am, [nested]]]
```

In addition, there is a very useful feature that allows us to split list into *Head and Tail*:

```
List = [1, 2, 3]
Tail = [2, 3]
-> List = .(1,Tail)
```

or in bracket notation:

```
List = [1|Tail]
```

In Prolog lists are handled as a binary trees, thus it accepts lists in different formats:

```
[1, 2, ...]
[Head|Tail]
[1, 2,...|Other]
```

1.6 Operators

We can define new operators by inserting certain kinds of clauses, called *directives*. For example, we may want to define the operator `has` for our rule `Alex has info.`, this can be done by defining the directive:

```
:- op(300, xfx, has).
```

where 300 is the precedence and `xfx` means the infix operator.

There are three groups of operators:

1. infix operators: `xfx, xfy, yfx`
2. prefix operators: `fx. fy`

3. postfix operators: `xf, yf`

where `f` represents the operator and `x` and `y` represent arguments.

1.7 Backtracking and Cut

Backtracking
We can also have backtracking in rules. For example consider the following program:

```
hold_party(X):-
   birthday(X),
   happy(X).

birthday(tom).
birthday(fred).
birthday(helen).

happy(mary).
happy(jane).
happy(helen).
```

If we now pose the query:

```
?- hold_party(Who).
```

In order to solve the above, Prolog first attempts to find a clause of birthday, it being the first subgoal of birthday. This binds X to tom. We then attempt the goal `happy(tom)`. This will fail, since it doesn't match t he above database. As a result, Prolog backtracks. This means that Prolog goes back to its last choice point and sees if there is an alternative solution. In this case, this means going back and attempting to find another clause of birthday. This time we can use clause two, binding X to fred. This then causes us to try the goal `happy(fred)`. Again this will fail to match our database. As a result, we backtrack again. This time we find clause three of birthday, and bind X to helen, and attempt the goal `happy(helen)`. This goal matches against clause 3 of our happy database. As a result, `hold_party` will succeed with `X=helen.`

Cut
Prolog automatically backtracks for us if this is necessary for satisfying the goal, although we may want to control that by preventing backtracking. We can do this by using `cut` function, or just `!`.

For me, the easiest way of understanding this concept is to use an example. Lets imagine that we have a relation `member(X, L).` which will check whether X is in list L. This program would look like:

```
member(X, [X | Xs]).
member(X, [Y | Ys]) :-
   member(X, Ys).
```

But such an implementation is not ideal, because if X occurs several times then any occurrence can be found. We can change `member` in a way such that the program will stop after the first occurrence of X:

```
member(X, [X | Xs]) :- !.
member(X, [Y | Ys]) :-
   member(X, Ys).
```

This program will generate just one solution, eg:

```
?- member(X, [1, 2, 3]).
   X = 1;
   no
```

I found the cut being very useful even for a simple rule like `not`:

```
not(A) :-
   A, !, fail ; true.
```

1.8 Practice

Using the basic concepts introduced so far let us write a few basic rules:

```
factorial:
fac(N) :- fac(N, 1).
fac(0, P) :- write(P).
fac(N, P) :- N>0, M is N-1, K is N * P, fac(M, K).

list concatenation:
conc([ ], L, L).
conc([ X | L1 ], L2, [ X | L3 ]) :-
   conc(L1, L2, L3).

calculation the length of the list:
length(L, N) :-
   length(L, 0, N).
length([], N, N).
length([_ | L], N0, N) :-
   N1 is N0+1,
   length(L, N1, N).

etc...
```

A lot more examples such as sudoku solver etc can be found on my github page `https://github.com/vitpro/ilp_alp_tal/tree/master/src`

2 Answer Set Programming

2.1 Basics and tools used

Answer Set Programming or *ASP* is a form of declarative programming based on stable model semantics of logic programming. The following tools were suggested:

gringo

Grounder capable of translating logic programs provided by the programmer into equivalent ground programs

clasp

Solver for the grounded programs computed by gringo, producing answer sets

clingo

Bounds `gringo` and `clasp`, taking a logic program as an input and producing answer sets an an output.

2.2 ASP features and comparison to Prolog

As in Prolog, syntax is very similar, as they both represent logic programming. Typical ASP program is a collection of:

- **Facts:** A_0.
- **Rules:** A_0 :- $L_0, ..., L_n$.
- **Integrity Constants:** :- $L_0, ..., L_n$.

Where A_0 is an *atom*, and any of L_i's are *literals*.
The set $\{L_0, ..., L_n\}$ is called the *body* of the rule. Facts have an empty body.

I will now give a brief description of some functionality which is mostly not featured in Prolog.

Negation

In ASP the predicate `not` can also be represented as `'-'`. Semantically, $-A$ will just be a new atom, where $A \cup -A \models \emptyset$.

Disjunction

Additionally, there is a disjunction connective `'|'` in ASP. Disjunction is true if at least one of its atoms is true. For example: `a | b.` will produce two answer sets $\{a\}$ and $\{b\}$. I understand it as *or*.

Assignments

Differently to Prolog, where assignments are implemented as an infix operator `is`, ASP provides us different notation of `:-`. There is also a comparison predicate `==`.
Worth mentioning that assignments *bind* variables, which excludes the cases containing looped assignments. For example, the code `r(A) :- A = B, B = A.` is rejected by `gringo`.

Intervals

There is a very neat approach to interval representations. This is easy to understand from an example: `number(1..9).` will mean all numbers from 1 to 9 inclusive.
Another handy feature is *pooling*, which means that instead of writing `rule(a). rule(b). rule(c).` we can write `rule(a;b;c).`

Conditions

Conditions can be represented by `':'` symbol. Lets consider the following program:

```
day(mon). day(tue). day(wed). day(thu). day(fri). day(sat). day(sun).
weekend(sat). weekend(sun).
that(X) : day(X) :- doSomething.
weekdays :- day(X) : day(X) : not weekend(X).
```

In my understanding, `':'` is somehow similar to `foreach` statements from most imperative programming languages. Lets consider the line `that(X) : day(X) :- doSomething.` Essentially, it just means `that(mon) | that(tue) | that(wed) | that(thu) | that(fri) | that(sat) | that(sun) :- doSomething.`
More interesting example is `weekdays :- day(X) : day(X) : not weekend(X).` allowing us to use the same atom, `day(X)`, twice. Gringo simplifies out code to this: `weekdays :- day(mon), day(tue), day(wed), day(thu), day(fri).` which is probably easier to read and understand.

Aggregates
ASP allows us to do operation on sets of literals that evaluates to some values. The aggregate has the following format: l `op` $[L_0 = \omega_0, ..., L_n = \omega_n]\upsilon$, where l and υ are lower and upper bounds respectively, operator `op` and a set of weighted literals L_i. Of course, the use of l and υ is entirely optional.
Here are a few examples using such aggregates:

```
1 #count {a,a, not b,not b} 1.
2 #sum [a, not b, c=2] 3
etc...
```

Note that `#count` requires curly brackets as there must be no weighted literals within an aggregate.

Meta Statements
In order to make the language more flexible there are meta statements introduced in ASP. There is a short list of them:

Comments

```
rule(1). % here is a comment
%*
commented_rule
*%
```

Hiding predicates

`#hide` is useful to hide information you don't want to display

```
#hide.              % Suppress all atoms in output
#hide p/3.          % Suppress all atoms of predicate p/3 in output
#hide p(X,Y) : q(X). % Supress p/3 if the condition holds
```

In order to selectively include the atoms of a certain predicate in the output, we can use the `#show` declarative.

```
#show p/3.            % Include all atoms of predicate p/3 in output
#show(X,Y) : q(X).    % Include p/3 if the condition holds
```

Use of constants

Constants are just place holders for concrete values provided by a user. By using the `#const` declarative we can define a default value to be inserted for a constant:

```
#const a = 15.
#const g = f(a,b).
```

Lua code snippets

Ability to include scripting language inside logic programs can be very useful, if used correctly (accessing the database, for example). It is done by writing `#begin_lua` and `#end_lua.`, and having your Lua code in between the tags. We can refer to our Lua snippet via `@` operator:

```
#begin_lua
function gcd(a, b)
  if a == 0 then return b
  else return gcd(b % a, a)
  end
end
#end_lua.

gcd(X,Y,@gcd(X,Y)) :- f(X,Y).
```

Examples

As for example I tried implementing TSP problem. From a small weighted graph consisting 4 nodes I got the following result:

```
vit@vitLaptop:~/programming/urop/src$ clingo -n 0 tsp.lp
Answer: 1
loop(4,3) loop(3,1) loop(2,4) loop(1,2)
SATISFIABLE
```

Code can be found on github.

3 Inductive Logic Programming

3.1 Basic Idea

Inductive Logic Programming, from here on referred to as *ILP*, is an approach to machine learning. In ILP the definitions of relations are deduced from examples, which means that a program is automatically constructed from a set of examples.
It is a framework that allows a programmer to define examples and some background knowledge instead of writing the programs directly.

Typical ILP problem formulation:
Given:

1. A set of positive E_+ and negative E_- examples
2. Background knowledge B, represented as a set of logic formulas, such that the examples E_+ cannot be derived from B

Find:
A hypothesis H, represented as a set of logic formulas, such that:

1. All examples in E_+ can be derived from B and H
2. No examples in E_- can be derived from B and H

More precisely:
An ILP task is defined as $\langle E, B, L_H \rangle$ *where* E *is a set of examples,* B *is a background knowledge and* L_H *set of logic theories, called language bias.*
H *is a hypothesis for a given ILP task* $\langle E, B, L_H \rangle$ iff H $\in L_H$, and B $\cup$ H $\models$ E.

3.2 HYPER

Idea
With some help I managed to develop a HYPER program (*Hyp*othesis refin*er*), which constructs Prolog programs through refinement of starting hypotheses.
This is done by searching a space of possible hypotheses and their refinements. The main idea is to take a hypothesis H_1 and produce a more specific hypothesis H_2 such that $H_2 \subset H_1$.
Such a set of hypotheses and their refinements is called a *refinement graph.*
Having built the graph the learning problem is reduced to searching this graph (searching from the over-general hypothesis to the most complete and consistent one).
An important point is that each successor of a hypothesis in the graph only covers the subset of the cases covered by the predecessor. This explains why during the search we only consider complete hypotheses, as an incomplete hypothesis cannot be refined into a complete one.

Search
We start the search with a set of start hypotheses. The refinement graph is considered to be a tree, and as the search starts with multiple start hypotheses, they become the roots of search trees. Therefore the search space we defined is just a set of trees. I have implemented best-first search by using a helper function that considers the size of a hypothesis and its accuracy with respect to an example set.

```
Cost(H) = 10*#literals(H) + #variables(H) + 10*#negativeEgs(H)
```

Full HYPER implementation can be found on github.

Examples
As an example I tried to implement insertion sort learning. The output produced:

```
Hypotheses generated: 3658
Hypotheses refined:    274
To be refined:         412
```

```
sort([],[]).
sort([A|B],D) :-
   sort(B,C),
   insert_sorted(A,C,D).
```

4 Abductive Logic Programming

ALP is a framework for ILP that is based on abductive reasoning. The solutions for the ALP task can be mapped back to solutions of the original ILP problem.
There are a lot of relations between abductive and inductive reasoning, where abduction is the reasoning process that derives possible explanations from observations(in other words, makes assumptions that would lead to solving the given goal). Induction, on the other hand, derives generalisations of a concept from given known rules.

ALP task
Here I will try to define ALP task in easy words. Let's consider the following program:

```
a :- b, c.
b :- i, j.
i :- c.
j.
```

Ans support we want to find `a`. We will run a query

```
?- a.
```

So the Prolog will try to find the solution by doing the following:

```
        ?- a.
           |
       ?- b, c.
           |
     ?- i, j, c.
           |
     ?- c, j, c.
           |
       ?- j, c.
```

At this point Prolog knows that `j` is true, but it has no `c`. ALP provides us with ability to assume it, and generate the set of such assumptions. In our case the set will just be `[c]`.

More formally: given a program Π *and a goal to reach* Γ*, then* Δ *is an abductive solution for the ALP task if* $\Pi \cup \Delta \models \Gamma$.

5 Top-direct Abductive Learning

Top-direct Abductive Learning, or *TAL* is also an ILP system, that builds on transformations.
Results from the transformations and abductive reasoning enables learning with negation as failure

as well as recursive learning. Also, this system guarantees completeness, as it is restricted by the underlying abductive system. Also, in TAL positive and negative examples are treated symmetrically.
With respect to other ILP tool TAL provides the following notable features:

- It supports non-monotonic learning
- Can be used to perform theory revision
- Uses configurable heuristics in the search
- Is complete, i.e. it is able to find a solution if one exists. This implies TAL is able to learn recursive and multi-predicate concepts, perform non-observational learning and predicate invention
- It is easily debuggable, providing a graphical representation of the search

TAL algorithm works the following way:
`Given`: B background theory, E_+ and E_- positive and negative examples, I integrity constrains and M mode declarations.
`Output`: hypothesis H.
`Algorithm`:
$T_M = PRE - PROCESSING(E, B, M)$
$\Delta = ABDUCE(T_M \cup B, I, E)$
$H = POST - PROCESSING(\Delta, M)$

Soundness and Completeness

Soundness. Let $\langle P, A, I\rangle$ *be an abductive program, and* $\langle \emptyset, \Delta, \Delta*, C\rangle$ *be a solution state for the query* q *with a* θ *substitution, then*
$P \cup \Delta \models q\theta$
and $P \cup \Delta$ *is consistent.*

Completeness. Let $\langle P, A, I\rangle$ *be an abductive program, and* W *be a derivation tree for a given query* q*. If* W *is finite then:*

- *if all the branches of* W *are failed, then* $P \models \bar{\forall}.(\neg q)$
- *if* $P \cup \bar{\exists}.q$ *is satisfiable, then* W *contains a successful branch.*

Remarks
If two solutions H' and H exists for the inductive task such that $H \subset H'$, H' is not guaranteed to be derived as it is not a subset-minimal solution. Although, this does not penalise the system in practice, as the additional rules in H' can be safely removed without affecting the coverage on the examples.

6 Simulators

In this section I will describe the practical aspect of my placement, as I have been proposed to work with traffic simulators and try out to plug real world data(such as tfl or openPath) in order to achieve something interesting(for example, predict what is the best place for the next London underground line, or help people plan their journeys more efficiently.