Declarative Programming

Main concepts & Exam questions

Bruno Rocha Pereira

Computer Science : SOFT

Vrije Universiteit Brussel August 14, 2016

Contents

Page

Main concepts

1.1 Clausal Logic

A logic system consists of:

1. **Syntax**: which sentences are legal
2. **Semantics**: what is the truth value of a sentence
3. **Proof theory**: how to derive new sentences (theorems) from assumed ones (axioms) by means of inference rules.

A logic system should be:

1. **Sound**: anything you can prove is true.
2. **Complete**: anything true can be proven.

There are different clausal logic systems (in order of increasing expressiveness):

1. **Propositional clausal logic**:

   Only composed by atoms with are statements with a value of true or false.

   e.g.: married; bachelor :- man, adult.

   The **Herbrand base** β_p is the set of all atoms occurring in the program.

   A **Herbrand interpretation** ***i*** is a mapping of all atoms in β_p to a truth value.

   An interpretation is a **model for a <u>clause</u>** if the clause is true under the interpretation.

   An interpretation is a **model for a <u>program</u>** if it is a model for each clause of the program.

2. **Relational clausal logic**:

   Introduces relations between constants and/or variables.

   e.g.: likes(Declarative, S) :- crazy(S).

   The **Herbrand universe** is the set of all atoms the set of all ground terms occurring in P.

   The **Herbrand base** β_p is the set of all ground atoms that can be constructed using predicates in P and arguments in the Herbrand universe.

   A **Herbrand interpretation** ***I*** is a subset of β_p, with all the ground atoms that are true.

3. **Full clausal logic**:

   To avoid explicit listing of clauses, we introduce function symbols and complex terms (*functors*)

   e.g.: loves(X, person_loved_by_(X)).

4. **Definite clausal logic**:

   This is what Prolog uses. Clauses only have one true litteral.

   e.g.: A :- B_1 , ... B_n

1.2 Cut

Once you reached me, stick with all variable substitutions you've found after you entered my clause.

Cuts are a procedural feature of Prolog and are expressed with the keyword !. When a cut is encountered, Prolog won't try alternatives for litterals left to the cut and the clause the cut is found, avoiding to backtrack.

1.2.1 Green Cut

Green cuts don't prune away success branches, it implies that the conjuncts on its left are deterministic and therefore don't need alternative solutions. It also implies that if the cut is reached, the clauses below with different head won't result in alternative solutions. Its use is to optimize the program.

e.g.:

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X), \+ gotmoney(X).
```

1.2.2 Red cut

Red cuts prune away success branches. This means that some logical consequences of the program are not returned. Changes the semantic of a program.

e.g.:

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X).
```

If for any reason the first rule is removed (e.g. by a cut-and-paste accident), the second rule will be broken, i.e., it will not guarantee the rule \+ (not) gotmoney(X).[1]

[1] (https://en.wikipedia.org/wiki/Cut_(logic_programming))

1.3 Negation as failure

In Prolog, the predicate not(X) is succeeds when X fails. It's implemented like this:

```
not(Goal) :- Goal, !, fail.
not(Goal).
```

1.4 Floundering

Floundering occurs when argument is not ground. The problem is that the implementation of the operator \+ only works when applied to a literal containing no variables, i.e., a ground literal. It is not able to generate bindings for variables, but only test whether subgoals succeed or fail. So to guarantee reasonable answers to queries to programs containing negation, the negation operator must be allowed to apply only to ground literals. If it is applied to a nonground literal, the program is said to flounder. [2].

For example,

```
\+ a(X), v(X).
```

will lead to floundering since X is not ground in the not predicate. Instead, in order to avoid it, we need to use:

```
v(X), \+ a(X).
```

1.5 Graphs

A prolog problem can be represented as a graph where the state is a `node`, the state transition is an `arc`, a `start node` and a `goal node` are defined and a solution is optimal if the cost over the path is minimal.

Search algorithms can be:

- **complete**: if a solution is always found if there is one.
- **optimal**: if the best solution can be found if there are several.
- **efficient**: if it respects memory and runtime requirements.
- **blind** of **informed**: if the quality of partial solutions steer the process. Blind happens in algorithms like **BSF**(Breath Search First) or **DFS**(Depth Search First).

[2] http://stackoverflow.com/questions/14715070/prolog-negation-and-logical-negation

1.6 Backward chaining & Forward chaining

1.6.1 Backward

Using backward chaining, the evaluation is done from head to body. The search starts where we want to be towards where we are. This is how Prolog query answering works.

1.6.2 Forward

Using backward chaining, the evaluation is done from body to head. The search starts where we are towards where we want to be. This is how a model construction works. This only workd for clauses where grounding the body also grounds the head (e.g.: **Best First Search**, that is based on a heuristic function.

Which one is the most effective depends on the structure of the search space.

1.7 Natural Language Processing

A Natural language is a language used for communication between humans. The syntax of a language is specified by a grammar, which is a set of grammar rules of the form:

```
Category1 --> Category2,Category3
Category2 --> [Terminal]
```

CategoryX Denotates a syntactic category.

Parsing a sentence and determine its grammatical structure. The outcome is a parse tree that shows the grammatical constituents. The opposite operation is called sentence generation.

1.8 Definite clause grammar

Context-sensitive grammars greatly increase the complexity of the parsing task; moreover, the grammatical structure of sentences cannot be simply described by a parse tree. We will restrict attention to context-free grammars, extended with some Prolog-specific features. The resulting grammars are called Definite Clause Grammars.

A grammar rule containing a terminal

```
verb --> [sleeps]
```

means: a list of words represents a verb if it is the list consisting of the single word ‘sleeps’.

Translated to Prolog:

```
verb([sleeps]).
```

For example, if we have:

```
sentence --> noun_phrase,verb_phrase.
noun_phrase --> proper_noun.
noun_phrase --> article,adjective,noun.
noun_phrase --> article,noun.
verb_phrase --> intransitive_verb.
verb_phrase --> transitive_verb,noun_phrase.
article --> [the].
adjective --> [lazy].
adjective --> [rapid].
proper_noun --> [achilles].
noun --> [turtle].
intransitive_verb --> [sleeps].
transitive_verb --> [beats].
```

We can generate a list of terminals like:

```
the lazy turtle sleeps
Achilles beats the turtle
the rapid turtle beats Achilles
```

DCG's are an excellent illustration of the power of declarative programming: specifying a grammar gives you the parser for free.

1.9 Reasoning with imcomplete information

1.9.1 Default Reasoning

Assumes the normal state of affairs, unless there is an evidence of the contrary.

```
If I push this button, the light in my room will be on.
```

If we have something like this:

```
Tweety is a bird.
Normally, birds fly.
Therefore, Tweety flies.
```

can be translated to

```
bird(tweety).
flies(X) :- bird(X), normal(X).
```

Except in default reasoning, we don't need to specify if things are normal. We only need to specify if the bird was abnormal.

We would then have:

```
bird(tweety).
flies(X) :- bird(X), not(abnormal(X)).
```

This is a non-monotonic form because new information can invalidate former conclusions, for example:

```
bird(tweety).
flies(X) :- bird(X), not(abnormal(X)).
ostrich(tweety).
abnormal(X) :- ostrich(X).
```

A form of reasoning is said to be **monotonic** when:

$$Theory|Conclusion => Theory \cup AnyFormula \mid Conclusion$$

This basically means than adding any new formulas won't change anything to the previously drawn conclusions.

1.9.2 Abductive Reasoning

Chooses between several explanations that explain an observation

```
The light doesn't switch on, the light bulb must be broken
```

Abducing is trying to prove **Observation** from theory when a literal is encountered that cannot be resolved (i.e. an abducible), add it to the **Explanation**. The basic idea is, having a head and a body, to try to prove the body. If it cannot be proved, it is added to the list of explanations.

$$Given\ a\ Theory\ T\ and\ an\ observation\ O,\ find\ an\ explanation\ E\ such\ that\ T\ \cup E\ \models O$$

1.10 Completing incomplete programs

$A\ program\ P\ is\ "complete"\ if\ for\ every\ (ground)\ fact\ f,\ either\ P \models f\ or\ P \models \neg f$

Completing an incomplete program is transforming it into a complete one that captures the intended meaning of the original program.

1.10.1 Closed world assumption

Everything that is not known to be true is false.

The first, simple method is called the Closed World Assumption; it is simple in the sense that it only works for definite clauses without negation. There is no need to say that something is false, as something not stated to be true is.

Thus, if P is a program and B_p is its Herbrand base, then we define the CWA-closure CWA(P) of P as:

$$CWA(P) = P \cup : -A | A \in B_p \wedge P \not\models A$$

where CWA(P) is the intended program according to the Closed World Assumption.

1.10.2 Predicate completion

The second method is called Predicate Completion; it can handle general programs with negated literals in the body of clauses. It may however lead to inconsistencies if the program is not stratified.

Its principle is to turn implications (if) into equivalences (iff) by completing clauses (with the "and-only-if" parts). E.g:

```
likes(peter,S):- student(S,peter).
```

Its completion is:

$$\forall X \forall\ likes(X,S) \leftrightarrow X = peter \wedge student(S, peter)$$

This gives in clausal form:

```
likes(peter,S):- student(S,peter). %Original clause
X=peter :-likes(X,S). % Added via the completion
student(S,peter) :- likes(X,S) % Added via the completion
```

The steps are:

1. Ensure that each argument of each clause head is a distinct variable
2. If there are several clauses for a predicate, combine them into a single formula.
3. Turn the implication into an equivalence.
4. Convert to clausal form.

This means that each clause nees to be seen as part of the definition of a specific predicate. The clause

```
likes(peter,S):- student_of(S,peter).
```

is seen as part of the definition of the `likes` predicate. Thus, X likes S if and only if X is Peter and S is a student of Peter.

1.10.3 Stable model

Guesses a model and verifies that it can be constructed from the program. It does thus introduce non-determinism.

1.11 Inductive Reasoning

Induction is a form of reasoning that generalizes a rule from a number of similar observations

```
It's dark, it must be after 9pm.
```

In induction, we have:

$$Theory \cup Hypothesis = Examples$$

Induction and **abduction** are quite similar. Their main difference is that in **Induction**, **Hypothesis** is allowed to be a set of clauses, while in **abduction** it has to be a set of ground facts. **Indtuction** reasoning is a hypothesis search involving successive **generalization** and **specialization** steps of a curent hypothesis. This means searching for a rule that generalizes those observations.

1.12 Unification and anti-unification

1.13 Bottom up induction & Top down induction

1.13.1 Bottom up

Constructs the **rlgg**(Relative Least General Generalization) of two positive examples. It first removes all positive examples that are extensionally covered by the constructed clause and further generalizes the clause by removing literals as long as no negative examples are covered.

1.13.2 Top down

Starts with the most general definition and further specializes the clause by adding literals as long as negative examples are covered