Discourse

BSc Artificial Intelligence, Spring 2011

Raquel Fernández

Institute for Logic, Language & Computation
University of Amsterdam

Summary from Last Week

Overview of the main topics of the course: from interpreting single sentences to processing discourse (pronouns, presuppositions, consistency, informativity,...)

We introduced the main ideas behind model-theoretic semantics: using logic as an intermediate level of representation to link language to the world.

To start with, we chose First Order Logic to represent natural language meaning.

- FO formulas ≈ formal meaning representations (truth conditions)
- FO models ≈ formal representations of world situations
- satisfaction relation ≈ knowing when a natural language sentence is true in a given situation (querying task)

We did a quick review of FOL.

Plan for Today

We would like to use FOL as a semantic representation formalism. How do we deal with it computationally with Prolog?

We shall address the querying task by implementing a first-order model checker:

- the model checker takes a first-order formula and a first-order model as input and checks whether the formula is true in the model.

We will first develop a preliminary version of a model checker and then refine it to fix some shortcomings.

A First-Order Model Checker in Prolog

Model Checker: Main Components

How should we implement a FO model checker in Prolog? There are three main tasks we have to deal with:

- Deciding how to represent first-order models.
- Deciding how to represent first-order formulas.
- Specifying how (representations of) formulas are to be evaluated in (representations of) models.

The latter task corresponds to implementing the satisfaction relation for first-order logic (given again in the next slide for convenience).

The Satisfaction Definition

The satisfaction relation holds between a formula, a model $D = (D, F)$, and an assignment g of values in D to variables.

[Here 'iff' is shorthand for 'if and only if.' That is, the relationship on the left-hand side holds precisely when the relationship on the right-hand side does too.]

1. $M, g \models R(\tau_1, \ldots, \tau_n)$ iff $(\mathcal{I}_F^g(\tau_1), \ldots, \mathcal{I}_F^g(\tau_n)) \in F(R)$;
2. $M, g \models \neg\varphi$ iff not $M, g \models \varphi$;
3. $M, g \models \varphi \wedge \psi$ iff $M, g \models \varphi$ and $M, g \models \psi$;
4. $M, g \models \varphi \vee \psi$ iff $M, g \models \varphi$ or $M, g \models \psi$;
5. $M, g \models \varphi \rightarrow \psi$ iff not $M, g \models \varphi$ or $M, g \models \psi$;
6. $M, g \models \forall x\varphi$ iff $M, g' \models \varphi$ for all x-variants g' of g; and
7. $M, g \models \exists x\varphi$ iff $M, g' \models \varphi$ for some x-variant g' of g.

Truth: A sentence φ is true in a model M iff for any assignment g, $M, g \models \varphi$. If φ is true in M, we write $M \models \varphi$.

Representing Models in Prolog (1)

Let's suppose that we have fixed our vocabulary. How should we represent models of this vocabulary in Prolog?

Vocabulary:

$$\{ \quad (\text{LOVE}, 2), (\text{CUSTOMER}, 1), (\text{ROBBER}, 1), (\text{JULES}, 0), (\text{VINCENT}, 0), (\text{PUMPKIN}, 0), (\text{HONEY-BUNNY}, 0), (\text{YOLANDA}, 0) \}$$

Prolog representation of a possible model:

```
model([d1,d2,d3,d4,d5],
      [f(0,jules,d1),
       f(0,vincent,d2),
       f(0,pumpkin,d3),
       f(0,honey_bunny,d4),
       f(0,yolanda,d5),
       f(1,customer,[d1,d2]),
       f(1,robber,[d3,d4]),
       f(2,love,[(d3,d4)])]).
```

- Recall that a FO model M is defined as an ordered pair (D, F)
- The predicate `model/2` mirrors this set-theoretical definition:
  - ∗ its first argument is a list representing the model's domain D, which in this case contains five elements `[d1,d2,d3,d4,d5]`
  - ∗ the second argument is a list specifying the interpretation function F

Representing Models in Prolog (2)

Example model representation 1:

```
model([d1,d2,d3,d4,d5],
      [f(0,jules,d1),
       f(0,vincent,d2),
       f(0,pumpkin,d3),
       f(0,honey_bunny,d4),
       f(0,yolanda,d5),
       f(1,customer,[d1,d2]),
       f(1,robber,[d3,d4]),
       f(2,love,[(d3,d4)])]).
```

Example model representation 2:

```
model([d1,d2,d3,d4,d5,d6],
      [f(0,jules,d1),
       f(0,vincent,d2),
       f(0,pumpkin,d3),
       f(0,honey_bunny,d4),
       f(0,yolanda,d4),
       f(1,customer,[d1,d2,d5,d6]),
       f(1,robber,[d3,d4]),
       f(2,love,[])]).
```

Example 1:

- The interpretation function names each of the elements in the domain.
- It also tells us that Jules and Vincent are customers, Pumpkin and Honey Bunny are robbers, and that Pumpkin loves Honey Bunny.

Representing Models in Prolog (3)

Example model representation 1:

```
model([d1,d2,d3,d4,d5],
      [f(0,jules,d1),
       f(0,vincent,d2),
       f(0,pumpkin,d3),
       f(0,honey_bunny,d4),
       f(0,yolanda,d5),
       f(1,customer,[d1,d2]),
       f(1,robber,[d3,d4]),
       f(2,love,[(d3,d4)])]).
```

Example model representation 2:

```
model([d1,d2,d3,d4,d5,d6],
      [f(0,jules,d1),
       f(0,vincent,d2),
       f(0,pumpkin,d3),
       f(0,honey_bunny,d4),
       f(0,yolanda,d4),
       f(1,customer,[d1,d2,d5,d6]),
       f(1,robber,[d3,d4]),
       f(2,love,[])]).
```

Example 2:

- Only four elements are named by constants; d_5 and d_6 are anonymous, but we know that they are customers.
- d_4 has two names, Yolanda and Honey Bunny.
- Also note that the two-place LOVE relation is empty.

All this is fine, as it is in the set-theoretic definition of FOL.

Representing Formulas in Prolog (1)

- To represent first-order formulas in Prolog, we need to:
  - ∗ decide how to represent FO variables
  - ∗ decide how to represent non-logical symbols
  - ∗ decide how to represent the logical symbols
- Variables: we will simply represent FO variables as Prolog variables.
- Non-logical symbols: FO constants c and relation symbols R will be represented by Prolog atoms `c` and `r`.

Given these conventions, we can now represent atomic formulas as Prolog terms. The equality symbol (a two-place relation) can be represented by the Prolog term `eq/2`.

$\textsc{love}(\textsc{vincent}, \textsc{mia})$	`love(vincent,mia)`
$\textsc{hate}(\textsc{butch}, x)$	`hate(butch,X)`
$\textsc{yolanda} = \textsc{honey-bunny}$	`eq(yolanda,honey_bunny)`

Representing Formulas in Prolog (2)

- Logical symbols: we will use the Prolog terms in (1) to represent the boolean connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$, and those in (2) to represent quantified formulas:

(1) boolean connectives: `and/2` `or/2` `imp/2` `not/1`

(2) quantified formulas: `all(X,Phi)` `some(X,Phi)`

Sample formulas:

```
and(love(vincent,mia), hate(butch,X))

imp(love(vincent,mia), not(hate(vincent,mia)))

all(X,hate(butch,X))
```

The Satisfaction Definition in Prolog

We now turn to the final task for implementing our model checker, specifying the satisfaction relation: how formulas are to be evaluated in models.

The predicate that carries out this task is called `satisfy/4`: `satisfy(Formula,Model,G,Pol)`. Its arguments are:

1. the formula to be tested;
2. the model;
3. an assignment function: a list of assignments of members of the model's domain to any free variables in the formula; and
4. a polarity feature (`pos` or `neg`) that tells us whether a formula should be positively or negatively evaluated.

We know how arguments 1 and 2 look like, but arguments 3 and 4 require further clarification.

The assignment function `G` in `satisfy/4`

- We shall use Prolog terms of the form `g(Variable,Value)` to indicate that a variable `Variable` has been assigned the element `Value` of the model's domain.
- The third argument of `satisfy/4` will thus be a list of terms of this form, one for each free variable in the formula.
- If the formula to be evaluated has no free variables (it's a sentence or closed formula), the list will be empty.

formula with a free variable:	`hate(butch,X)`	`G = [g(X,d3)]`
sentence or closed formula:	`all(X, hate(butch,X))`	`G = []`

The `Polarity` argument in `satisfy/4` (1)

- When we evaluate a formula, we use the satisfaction definition to break it down into smaller subformulas, and then check these smaller subformulas in the model.
- When we encounter a negation, that tells us that what follows has to be checked as *false* in the model.
- Going deeper into a negated formula, we may encounter another negation, which means that its argument has to be evaluated as *true*. And so on and so forth...

$\textsc{robber}(\textsc{yolanda})$

```
robber(yolanda)
```

$\neg\textsc{robber}(\textsc{yolanda})$

```
not(robber(yolanda))
```

$\neg(\textsc{robber}(\textsc{yolanda}) \vee \neg(\textsc{yolanda} = \textsc{honey-bunny}))$

```
not(and(robber(yolanda), not(eq(yolanda,honey_bunny))))
```

The `Polarity` argument in `satisfy/4` (2)

- The polarity argument is a flag that records whether we are trying to check if a particular formula is true or false in a model.
  - ∗ subformula flagged as `pos`: check if it is true
  - ∗ subformula flagged as `neg`: check if it is false
- Note that when we give the original formula to the model checker, the polarity argument will be `pos`.

The heart of the model checker is a series of clauses that spell out recursively how to check a formula as true in a model, and how to check it as false.

Clauses of `satisfy/4`: Negation

Let's start by looking into how to handle complex formulas.

Negation:

```
satisfy(not(Formula),Model,G,pos):-
   satisfy(Formula,Model,G,neg).

satisfy(not(Formula),Model,G,neg):-
   satisfy(Formula,Model,G,pos).
```

- These clauses spell out the required flip-flops between true and false with negation.
- First clause: to check if a negated formula (`not(Formula)`) is true (`pos`), we need to check if the argument of the negation (`Formula`) is false (`neg`).
- Second clause: to check if `not(Formula)` is false (`neg`), we need to check if the argument of the negation (`Formula`) is true (`pos`).

Clauses of `satisfy/4`: Conjunction & Disjunction

Conjunction:

```
satisfy(and(Formula1,Formula2),Model,G,pos):-
   satisfy(Formula1,Model,G,pos),
   satisfy(Formula2,Model,G,pos).

satisfy(and(Formula1,Formula2),Model,G,neg):-
   satisfy(Formula1,Model,G,neg);
   satisfy(Formula2,Model,G,neg).
```

Disjunction:

```
satisfy(or(Formula1,Formula2),Model,G,pos):-
   satisfy(Formula1,Model,G,pos);
   satisfy(Formula2,Model,G,pos).

satisfy(or(Formula1,Formula2),Model,G,neg):-
   satisfy(Formula1,Model,G,neg),
   satisfy(Formula2,Model,G,neg).
```

These are direct Prolog encodings of what the satisfaction definition tells us about $\wedge$ and $\vee$. Note the use of the ; symbol, Prolog's built in *'or'*.

Clauses of `satisfy/4`: Implication

Implication:

Recall the formal satisfaction definition for this connective:

$M, g \models \varphi \to \psi$ iff not $M, g \models \varphi$ or $M, g \models \psi$;

The connective $\to$ can be defined in terms of $\neg$ and $\vee$:

- for any formulas ϕ and ψ whatsoever, $\phi \to \psi$ is equivalent to $\neg\phi \vee \psi$

Since we already have the clauses for $\neg$ and $\vee$ at our disposal, to check an implication we will simply check the equivalent formula:

```
satisfy(imp(Formula1,Formula2),Model,G,Pol):-
  satisfy(or(not(Formula1),Formula2),Model,G,Pol).
```

Clauses of `satisfy/4`: Quantifiers (1)

Existential quantification:

```
satisfy(some(X,Formula),model(D,F),G,pos):-
   memberList(V,D),
   satisfy(Formula,model(D,F),[g(X,V)|G],pos).

satisfy(some(X,Formula),model(D,F),G,neg):-
   setof(V,
         (memberList(V,D),satisfy(Formula,model(D,F),[g(X,V)|G],neg)),
         Dom),
   setof(V,memberList(V,D),Dom).
```

[N.B.: the predicate `memberList/2` succeeds if its first argument (any term) is a member of its second argument (a list). You can find it in the library `comsemPredicates.pl`. It's equivalent to the predicate usually called `member/2`.]

Clauses of satisfy/4: Quantifiers (2)

Let's examine the first clause in the code for existential quantification:

```
satisfy(some(X,Formula),model(D,F),G,pos):-
   memberList(V,D),
   satisfy(Formula,model(D,F),[g(X,V)|G],pos).
```

Recall the formal definition:

$$M, g \models \exists x \varphi \text{ iff } M, g' \models \varphi \text{ for some } x\text{-variant } g' \text{ of } g$$

$\exists x \varphi$ is true in a model M with respect to an assignment g if there is an x-variant assignment g' under which φ is true in M.

- the call `memberList(V,D)` instantiates the variable `V` to some element in the domain `D`
- with the instruction `[g(X,V)|G]` we try to evaluate with respect to this variant assignment
- if this fails, Prolog will backtrack and `memberList(V,D)` will give us another instantiation (if possible)
- we use Prolog backtracking to try out different variant assignments

Clauses of `satisfy/4`: Quantifiers (3)

Let's now examine the second clause:

```
satisfy(some(X,Formula),model(D,F),G,neg):-
   setof(V,
         (memberList(V,D),satisfy(Formula,model(D,F),[g(X,V)|G],neg)),
         Dom),
   setof(V,memberList(V,D),Dom).
```

The formula needs to be false with respect to all variant assignments:

- like before, we use `memberList(V,D)` and backtracking to try different variant assignments
- we use Prolog inbuilt `setof/3` predicate to collect all the instantiations of `V` that falsify the formula
- when all instantiations make the formula false, then `setof/3` will simply be the domain `D` itself
- obtaining `D` as the result of `setof/3` is the signal that we have falsified the existential formula.

Clauses of `satisfy/4`: Quantifiers (4)

Universal quantification:

Recall the formal definition:

$M, g \models \forall x \varphi$ iff $M, g' \models \varphi$ for all x-variants g' of g;

$\forall x \phi$ is logically equivalent to $\neg \exists x \neg \phi$, so we can reuse our existing code:

```
satisfy(all(X,Formula),Model,G,Pol):-
   satisfy(not(some(X,not(Formula))),Model,G,Pol).
```

Clauses of `satisfy/4`: Atomic Formulas

Let's now turn to atomic formulas. Recall the formal definition:

$M, g \models R(\tau_1, \ldots, \tau_n)$ iff $(\mathcal{I}_F^g(\tau_1), \ldots, \mathcal{I}_F^g(\tau_n)) \in F(R)$;

One-place predicate symbols (positive version):

```
satisfy(Formula,model(D,F),G,pos):-
   compose(Formula,Symbol,[Argument]),
   i(Argument,model(D,F),G,Value),
   memberList(f(1,Symbol,Values),F),
   memberList(Value,Values).
```

Remarks:

- we've again made used of `memberList/2`
- we've also used `compose/3` defined as (see `comsemPredicates.pl`):

```
compose(Term,Symbol,ArgList):-
    Term =.. [Symbol|ArgList].
```

  we use it to *decompose* a formula: it flattens a term into a list, with the predicate as the first member and the arguments as the rest.

Clauses of `satisfy/4`: Atomic Formulas (1)

Recall the formal definition of the interpretation of atomic formulas:

$M, g \models R(\tau_1, \ldots, \tau_n)$ iff $(\mathcal{I}_F^g(\tau_1), \ldots, \mathcal{I}_F^g(\tau_n)) \in F(R)$;

One-place predicate symbols:

```
satisfy(Formula,model(D,F),G,pos):-
   compose(Formula,Symbol,[Argument]),
   i(Argument,model(D,F),G,Value),
   memberList(f(1,Symbol,Values),F),
   memberList(Value,Values).

satisfy(Formula,model(D,F),G,neg):-
   compose(Formula,Symbol,[Argument]),
   i(Argument,model(D,F),G,Value),
   memberList(f(1,Symbol,Values),F),
   \+ memberList(Value,Values).
```

The real work is done by predicate `i/4`, which is a Prolog version of the interpretation function $\mathcal{I}_F^g$.

Clauses of `satisfy/4`: Atomic Formulas (2)

Recall from our formal definition that $\mathcal{I}_F^g$ interprets variables with assignment g and constants with the interpretation function F:

- if τ is a constant, $\mathcal{I}_F^g(\tau) = F(\tau)$
- if τ is a variable, $\mathcal{I}_F^g(\tau) = g(\tau)$

This is exactly what `i/4` does:

```
i(X,model(_,F),G,Value):-
   (
       var(X),
       memberList(g(Y,Value),G),
       Y==X, !
   ;
       atom(X),
       memberList(f(0,X,Value),F)
   ).
```

Clauses of `satisfy/4`: Atomic Formulas (3)

One-place predicate symbols: how are they handled?

```
satisfy(Formula,model(D,F),G,pos):-
   compose(Formula,Symbol,[Argument]),
   i(Argument,model(D,F),G,Value),
   memberList(f(1,Symbol,Values),F),
   memberList(Value,Values).

satisfy(Formula,model(D,F),G,neg):-
   compose(Formula,Symbol,[Argument]),
   i(Argument,model(D,F),G,Value),
   memberList(f(1,Symbol,Values),F),
   \+ memberList(Value,Values).
```

- `compose/3` turns the formula into a list
- `i/4` is called to interpret the argument term
- the 1st call to `memberList/2` looks up the meaning of the 1-place predicate
- the second call checks that the interpretation of the term is one of the possible values for the predicate in the model
- in the negative clause (`neg`), we use negation as failure `\+ memberList(Value,Values)` to check precisely the opposite

Clauses of `satisfy/4`: Atomic Formulas (4)

Two-place predicate symbols: for two-place predicate symbols, we make to calls to `i/4`, one for each of the two argument terms.

```
satisfy(Formula,model(D,F),G,pos):-
   compose(Formula,Symbol,[Arg1,Arg2]),
   i(Arg1,model(D,F),G,Value1),
   i(Arg2,model(D,F),G,Value2),
   memberList(f(2,Symbol,Values),F),
   memberList((Value1,Value2),Values).

satisfy(Formula,model(D,F),G,neg):-
   compose(Formula,Symbol,[Arg1,Arg2]),
   i(Arg1,model(D,F),G,Value1),
   i(Arg2,model(D,F),G,Value2),
   memberList(f(2,Symbol,Values),F),
   \+ memberList((Value1,Value2),Values).
```

Finally, these are the clauses for equality:

```
satisfy(eq(X,Y),Model,G,pos):-
   i(X,Model,G,Value1),
   i(Y,Model,G,Value2),
   Value1=Value2.
```

```
satisfy(eq(X,Y),Model,G,neg):-
   i(X,Model,G,Value1),
   i(Y,Model,G,Value2),
   \+ Value1=Value2.
```

Evaluating Formulas

Now we have the core of (the first version of) our model checker.

In order to make it more usable for handling the querying task, the code includes the predicate `evaluate/3`:

- it takes a formula, a numbered example model, and a list of assignments to free variables;
- it then checks whether the formula is satisfied in the example model and prints the result.

```
evaluate(Formula,Example,Assignment):-
   example(Example,Model),
   satisfy(Formula,Model,Assignment,Result),
   printStatus(Result).
```

For closed formulas, you can also use `evaluate/2`, which evaluates with respect to the empty list of assignments.

Beyond the Querying Task

The model checker can handle the querying task, which corresponds to the speakers' knowledge of whether a natural language sentence is true or false in a given situation.

But it can actually do more than that: it can be used to find elements in a model's domain that satisfy certain descriptions.

```
?- satisfy(robber(X),
           model([d1,d2,d3],
              [f(0,jules,d1),
               f(0,vincent,d2),
               f(0,pumpkin,d3),
               f(1,customer,[d2]),
               f(1,robber,[d1,d3]),
               f(2,love,[])]),
           [g(X,Value)],
           pos).

Value = d1 ;
Value = d3 ;
fail.
```

Note the assignment `[g(X,Value)]` where `Value` is a Prolog variable.

Test Suite

The predicate `modelCheckerTestSuite/0` can be used to test the model checker on predefined examples:

```
?- modelCheckerTestSuite.
```

This command will force Prolog to evaluate all the examples in the file `modelCheckerTestSuite.pl` and print out the results. The entries in that file have the following form:

```
test(some(X,robber(X)),1,[],pos).
```

The output of `modelCheckerTestSuite/0` will be a list of entries such as:

```
 Input formula:
 1 some(A, some(B, love(A, B)))
 Example Model: 1
 Status: Satisfied in model.
 Model Checker says: Satisfied in model.
```

The fourth line should the actual status of the formula, and the bottom line shows the verdict of the model checker.

Summary of programs for the Model Checker v.1

All programs can be found in the directory BB1, available from
`http://homepages.inf.ed.ac.uk/jbos/comsem/software1.html`

- `modelChecker1.pl` is the main file. Consult it to run the first-order model checker – it will load all other files it requires.
- `comsemPredicates.pl` contains definitions of auxiliary predicates, such as `memberList/2` and `compose/3`.
- `exampleModels.pl` contains example models in various vocabularies; you could add your own models to this file.
- `modelCheckerTestSuite.pl` contains formulas to be evaluated in the example models.

Problems with Model Checker v.1 (1)

Although the first version of our model checker implements the main ideas of the first-order satisfaction definition, it is not sufficiently robust.

These are the weak points — you are encouraged to think your way through them by yourself (e.g. by using the trace mode):

- Problem 1: evaluating Prolog variables

The model checker v.1 gets into an infinite loop when given the following queries:

```
?- evaluate(X,1).
?- evaluate(imp(X,Y),4).
```

- Problem 2: evaluating constants (i.e. terms instead of formulas)

The model checker should signal that terms can't be evaluated, but it doesn't:

```
?- evaluate(mia,3).
fail.

?- evaluate(all(mia,vincent),2).
fail.
```

Problems with Model Checker v.1 (2)

- Problem 3: evaluating out-of-vocabulary formulas

The model checker should signal that the satisfaction relation should not be defined for items that don't belong to the vocabulary, but it doesn't:

```
?- evaluate(tasty(cheese),1).
fail.
```

- Problem 4: evaluating formulas with free variables with `evaluate/2`

The model checker should signal that formulas with free variables can't be evaluated without an assignment, but it doesn't:

```
?- evaluate(customer(X),1).
fail.
```

A Refined Model Checker

The code in `modelChecker2.pl` implements an improved version of the model checker that handles these problems and produces appropriate messages.

We won't discuss the code for this revised version. You are encouraged to work out by yourself how it fixes the problems.

See the homework exercises for this week.

Next Week

So far we have been dealing with the querying task assuming that sentences in natural language can be translated into FOL formulas.

But - how do we actually associate sentences in plain natural language with logical semantics representations?

Next week we will look into the task of semantic construction:

- overview of how the lambda calculus can be used for this purpose;
- implementation of the lambda calculus in Prolog.

Some relevant code for next week:

```
experiment3.pl                lambda.pl
betaConversion.pl             readLine.pl
alphaConversion.pl            sentenceTestSuite.pl
betaConversionTestSuite.pl    semLexLambda.pl
comsemPredicates.pl           semRulesLambda.pl
```