Lecture 11 Demonstrating FOL Reasoning

COMP24412: Symbolic AI

Giles Reger

March 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Look at things required to make the previous theory work (in Vampire)

Learning Outcomes

By the end of this lecture you will be able to:

1. Order terms and literals using KBO
2. Apply the given clause algorithm
3. (For the lab) run Vampire

First-Order Logic Stuff

Syntax (propositional logic with predicates and quantifiers)

Semantics in terms of models

Clausal representation and reasoning with Clauses using Resolution

Reasoning with Equality with Paramodulation (and Equality Resolution)

Transformation to Clausal Form

Completeness via Model Construction

Ordered Resolution and clause selection (more today)

Important: naming conventions

On Friday I was asked how you can tell what are variables and what are constants

I will always use letters from the end of the alphabet (u,v,w,x,y,z) for variables and letters from the start of the alphabet (a,b,c,d) for constants. I'll also tend to use f,g,h for function symbols, p,q,r for predicates, and s,t for terms. I will sometimes use *sk* for Skolem constants/functions.

If you prefer you can use capital letters X,Y for variables as in Prolog.

Note that in inference rules we have *meta-variables* e.g. they are templates

The Hidden Rule

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

The Hidden Rule

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

The Hidden Rule

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

But they represent $(\forall x.p(x, a)) \wedge (\forall x.p(b, x))$, which is unsat

The Hidden Rule

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

But they represent $(\forall x.p(x, a)) \wedge (\forall x.p(b, x))$, which is unsat

Equivalent to $(\forall x.p(x, a)) \wedge (\forall y.p(b, y))$ i.e. clauses $p(x, a)$ and $\neg p(b, y)$

Hidden rule: Naming Apart.
When unifying literals from two different clauses you should first rename variables so that the literals do not share literals.

Sometimes I do this implicitly e.g. in $p(x)$ and $\neg p(x)$ we should rename to $p(x)$ and $\neg p(y)$ and then apply $\{x \mapsto y\}$ but that's tedious.

What Non-Ground Clauses Mean

It is helpful to get an intuition for this. What does

$$p(x, y) \rightarrow q(x, z)$$

mean?

We can think of non-ground clauses as representing all instances of that clause. This is probably infinite.

When we reason with non-ground clauses we reason with the those sets.

We usually need to instantiation the ground-clause a bit first so that we are reasoning with the sets that we want. All non-ground inference rules can be split into two phases: instantiation and then 'ground' reasoning.

Ordered Resolution

On Friday I introduced ordered ground resolution

Lifting this to the non-ground case requires us to be a bit more explicit about orderings

Ground Term Ordering

Let the number of (function or variable) symbols in a term be its weight given by a function w e.g. $w(f(a, g(x))) = 4$.

Let $\succ_S$ be an ordering on function symbols. We define a well-founded total ordering on ground terms such that $s \succ_G t$ if

1. $w(s) > w(t)$, or
2. $w(s) = w(t)$ and $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ and either
   - $f \succ_S g$, or
   - $s_i \succ_G t_i$ for some i and for all $j < i$, $s_j = t_j$

Why is this total?

Ground Term Ordering

Let the number of (function or variable) symbols in a term be its weight given by a function w e.g. $w(f(a, g(x))) = 4$.

Let $\succ_S$ be an ordering on function symbols. We define a well-founded total ordering on ground terms such that $s \succ_G t$ if

1. $w(s) > w(t)$, or
2. $w(s) = w(t)$ and $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ and either
   - $f \succ_S g$, or
   - $s_i \succ_G t_i$ for some i and for all $j < i$, $s_j = t_j$

Why is this total?

Examples (given $f \succ_S g \succ_S h \succ_G a \succ_G b$):

$$f(a) \succ_G a \qquad f(a) \succ_G h(a) \qquad g(a, f(a)) \succ_G g(a, f(b))$$

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)?

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)? Consider $f(g(a), x)$ and $f(x, x)$.
Why is this partial?

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)? Consider $f(g(a), x)$ and $f(x, x)$.
Why is this partial? Consider $f(x)$ and $f(b)$ where $a \succ_S b \succ_S c$

Examples (given $f \succ_S g \succ_S a$):

$$f(x) \succ_N g(x) \qquad f(x) \succ_N x \qquad g(x, f(y)) \succ_N g(x, a)$$

What about $g(x, y)$ and $g(y, x)$?

This ordering is $\succ_{KBO}$ the Knuth-Bendix Ordering (KBO) used in Vampire

Ordered Resolution

$\succ_{KBO}$ lifts to predicates directly (extend $\succ_S$ to predicate symbols).

Lift $\succ_{KBO}$ to literals: $\neg l \succ l$ for every atom l, treat $=$ as biggest predicate.

Need a slightly different characterisation of ordered resolution using selection functions - we perform resolution on selected literals.

Ordered Resolution is then

$$\frac{l_1 \vee C \qquad \neg l_2 \vee D}{(C \vee D)\theta}\ \theta = \mathrm{mgu}(l_1, l_2)$$

where l_1 and $\neg l_2$ are selected.

A selection function is well-behaved if it either selects (i) at least one negative literal, or (ii) all maximal literals. This ensures completeness.

Fairness: Clause Selection

Firstly, we want to ensure that every clause is eventually selected

Secondly, we may want to select 'good' clauses earlier

A saturation process is fair if no clause is delayed infinitely often

Two fair clause selection strategies:

- First-in first-out
- Smallest (in number of symbols, e.g. weight) first
  (there are a finite number of terms with at most k symbols)

Vampire chooses between the two with a given ratio

Given Clause Algorithm

input: *Init*: set of clauses;
var *active*, *passive*, *unprocessed*: set of clauses;
var *given*, *new*: clause;
active := $\emptyset$; *unprocessed* := *Init*;
loop
  while *unprocessed* $\neq \emptyset$
    new :=*pop*(*unprocessed*);
    if *new* $= \square$ **then return** *unsatisfiable*;
    add *new* to *passive*
  if *passive* $= \emptyset$ **then return** *satisfiable* or *unknown*
  given := *select*(*passive*); *(* clause selection *)*
  move *given* from *passive* to *active*;
  unprocessed :=*infer*(*given*, *active*); *(* generating inferences *)*

Example 1

$$\left\{ \begin{array}{c} \forall x.(happy(x) \leftrightarrow \exists y.(loves(x,y))) \\ \forall x.(rich(x) \rightarrow loves(x, money)) \\ rich(giles) \end{array} \right\} \models happy(giles)$$

Example 2

$$\left\{ \begin{array}{c} \forall x.(require(x) \rightarrow require(depend(x))) \\ depend(a) = b \\ depend(b) = c \\ require(a) \end{array} \right\} \models require(c)$$

Vampire

Automated theorem prover for first-order logic

Implements everything we've talked about (and more)

Very efficient/powerful (wins lots of competitions)

Used as a back-box solver in lots of other things (in academia and industry)

Available from `https://vprover.github.io`

You have to use Vampire in lab 2b

TPTP

Language for describing first-order formulas in ASCII format.

See `http://tptp.cs.miami.edu/~tptp/`

For example

```
fof(one,axiom, ![X] : (happy(X) <=> (?[Y] : loves(X,Y)))).
fof(two,axiom, ![X] : (rich(X) => loves(X,money))).
fof(three,axiom, rich(giles)).
fof(goal,conjecture, happy(giles)).
```

Important - `axiom` for knowledge base, `conjecture` for goal and Vampire will do the negation for you.

Run Vampire

Vampire is a command-line tool

Run using

```
./vampire <options> <file>
```

I recommend setting certain options to make Vampire behave more similarly to the things defined in this course

```
-updr off -fde none -nm 0 -av off -fsr off -s 1 -sa discount
```

First 3 options are preprocessing optimisations, last 4 are proof search options.

Vampire

Lots of extra bits not in this course:

- Superposition (ordered paramodulation)
- Simplifications (saturation up to redundancy)
- Clause splitting
- Finite model building
- Arithmetic and datatype reasoning
- Lots and lots of proof search heuristics

If you're interested, especially if you'd like to do a 3rd year project in this area, then come and chat to me about it