COMP24412
Academic Session: 2018-19

Lab Exercise 2B: First-Order Reasoning with Vampire

For this lab exercise you should do all your work in your COMP24412/ex2 directory.

This lab exercise extends what you did in lab 2a. You will submit both parts together.

Learning Objectives

At the end of this lab you should be able to:

1. Translate first-order formulas into the TPTP format
2. Run Vampire on TPTP problems and explain what the proofs mean
3. Relate Vampire options to the given-clause algorithm given in lectures
4. Use Vampire to answer queries on a 'real' knowledge base

Getting Vampire

Before you can begin you'll need to grab a version of Vampire. I've put a pre-compiled binary in `/opt/info/courses/COMP24412` but if you want to build your own (e.g. on your own machine) then you can check out the tool from `https://vprover.github.io/download.html` and build it yourself.

Getting More Information

Hopefully this lab manual is reasonably self contained but if you want more information about the things mentioned in it then here are some places to look:

- There is an Academic Paper from 2013 that gives a good theoretical overview of Vampire and how it works
- I gave a day tutorial on Vampire in 2017 and the material can be found here although it is not very well organised at the moment.
- There is a lot more information about TPTP at their website and information about other solvers can be found at the competition website

Part 0: Solving A Simple Problem with Vampire

This part is not assessed but you should complete it as the rest of the lab will not make much sense without it. The aim here is to run Vampire on a few small problems and in doing so learn

1. How to represent first-order formulas in the TPTP syntax
2. How to run Vampire form the command line
3. How to read Vampire proofs

Writing Problems in TPTP

The first thing we're going to try and solve in Vampire is the following small example problem from lectures. We want to check the following entailment.

$$\begin{array}{l} \forall x.(rich(x) \rightarrow happy(x) \\ rich(giles) \end{array} \models happy(giles)$$

The first step is to write this problem down in a format that Vampire can read. The default[1] input format for Vampire is TPTP, which is almost exactly an ACSII version of the syntax we're used to. The above problem will be written as follows in TPTP

```
fof(one,axiom, ![X] : (rich(X) => happy(X))).
fof(two,axiom, rich(giles)).
fof(three, conjecture, happy(giles)).
```

This consists of three formulas where a formula is of the form `form(name,kind,formula)` where

- `form` describes the logic being used. Here we will always use `fof` standing for *first-order form* or `cnf` standing for *conjunctive normal form*.
- `name` is a name given to the formula. Vampire only uses this in proofs and ignores it the rest of the time.
- `kind` describes the kind of formula. We will mainly use `axiom` and `conjecture` although later you will see us using `question` as well here. The meaning of a TPTP file is that the conjunction of the axiom formulas entails the conjecture formula. You can have at most one conjecture. You can have no conjecture and then you are checking the consistency of the axioms.
- `formula` is the logical formula being described, we discuss this further below.

The syntax of formulas is what one might expect for an ASCII based formula representation. It is important to note that TPTP copies Prolog by using words beginning with upper-case letters for variables and those beginning with lower-case letters for constant/function/predicate symbols. Briefly the relevant syntax is:

- `~` for not
- `&` for conjunction (and), `|` for disjunction (or)
- `=>` for implication, `<=>` for equivalence
- `![X] :` for universal quantification. Note that you can group these e.g. `![X,Y] :`
- `?[X] :` for existential quantification
- `=` for equality and `!=` for disequality

[1] Vampire also accepts SMT-LIB format, which is a lisp-based syntax used by SMT solvers.

Now that you know the syntax try writing out the following formula in TPTP syntax

$$\forall x.((\exists y.p(x,y)) \rightarrow q(f(x)))$$

if that was tricky then go back and read the previous explanation again or ask for help. You can check whether your solution is syntatically correct by running `./vampire --mode ouptut problem.p` where you put your TPTP problem in `problem.p`.

Take all/some of the FOL formulas you wrote in Part 2 of the lab exercise 2A and translate them into the TPTP representation.

Proving things with Vampire

Save the above TPTP problem for the `rich(X) => happy(X)` example in `rich.p`. The first thing Vampire will do is turn the problem into a set of clauses. We can see how Vampire does this by running the following command

```
./vampire --mode clausify rich.p
```

does the result make sense? Can you write out the resulting clauses in English and in the first-order logic syntax we are used to.
Now let's solve the problem by running

```
./vampire rich.p
```

The proof shows the full clausification process and then the reasoning steps. The clausification process contains a `flattening` step, this is just an intermediate rewriting step that Vampire often performs - here it was unnecessary. The proof consists of two reasoning steps labelled `resolution` and `subsumption resolution`. This last rule is just a special kind of resolution where we also do some optimisation to the search space that we don't see in the proof (we delete one of the parents of the resolution). You can think of `submsumption resolution` just as `resolution`. However, the `submsumption resolution` rule is applied at a different point in proof search than `resolution` so let's turn it off for now with the option `-fsr off`.

Now try running

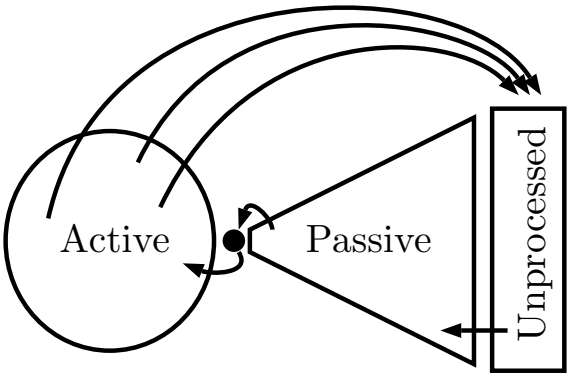
```
./vampire -fsr off --selection 0 rich.p
```

Do you get a different proof? You should because we have changed how we are doing literal selection. The value 0 just means select everything so we get unordered resolution in this case. If you run

```
./vampire -fsr off --selection 0 rich.p --show_active on --show_new on
```

you will see the order in which clauses are selected and which clauses are produced by activating (selecting) them.

Updated. *Reminder about given clause algorithm.* Vampire runs a given-clause algorithm (as discussed in the lectures). Remember that this works by iteratively selecting a clause and adding it to a set of active (selected) clauses and performing all inferences between the selected clause and the active set. This is illustrated in the below diagram where we have an active set, a passive set, and an unprocessed set (as we did in the algorithm shown in lectures). When we do `--show_active on` we are asking Vampire to print out the clauses it selects for activation. When we do `--show_new on` we are asking Vampire to print out all of the clauses produced by inferences.



To avoid doubt, we will reproduce what you should have found here. Without setting selection to 0 you should get the following new and active clauses

```
[SA] new: 7. happy(X0) | ~rich(X0) (0:4) I(0) [cnf transformation 6]
[SA] new: 8. rich(giles) (0:2) I(0) [cnf transformation 2]
[SA] new: 9. ~happy(giles) (0:2:G) I(0) [cnf transformation 5]
[SA] active: 9. ~happy(giles) (0:2:1:G) I(0) [cnf transformation 5]
[SA] active: 8. rich(giles) (0:2:1) I(0) [cnf transformation 2]
[SA] active: 7. ~rich(X0) | happy(X0) (0:4:1) I(0) [cnf transformation 6]
[SA] new: 10. happy(giles) (1:2) I(0) [resolution 7,8]
[SA] active: 10. happy(giles) (1:2:1) I(0) [resolution 7,8]
[SA] new: 11. $false (2:0:G) I(0) [resolution 10,9]
```

and with selection as 0 you should get

```
[SA] new: 7. happy(X0) | ~rich(X0) (0:4) I(0) [cnf transformation 6]
[SA] new: 8. rich(giles) (0:2) I(0) [cnf transformation 2]
[SA] new: 9. ~happy(giles) (0:2:G) I(0) [cnf transformation 5]
[SA] active: 9. ~happy(giles) (0:2:1:G) I(0) [cnf transformation 5]
[SA] active: 8. rich(giles) (0:2:1) I(0) [cnf transformation 2]
[SA] active: 7. happy(X0) | ~rich(X0) (0:4:2) I(0) [cnf transformation 6]
[SA] new: 10. ~rich(giles) (1:2:G) I(0) [resolution 7,9]
[SA] new: 11. happy(giles) (1:2) I(0) [resolution 7,8]
[SA] active: 10. ~rich(giles) (1:2:1:G) I(0) [resolution 7,9]
[SA] new: 12. $false (2:0:G) I(0) [resolution 10,8]
```

What's the difference? Can you explain what's happening. Note that this is *ordered* resolution being performed and without setting selection to 0 we use a custom selection function.

Let me briefly explain what we mean by the line

```
[SA] active: 10. ~rich(giles) (1:2:1:G) I(0) [resolution 7,9]
```

The part `[SA] active:` means that this is coming from the Saturation Algorithm and is about clause activation (when a clause is selected in the given clause algorithm). The `10` is the number of the clause - clauses are numbered as they are produced. Then we have the clause `~rich(giles` in cnf format. The string `(1:2:1:G) I(0)` gives the age (1), the weight (2) and the number of selected literals (1), which are always the leftmost literals in active clauses. The `I(0)` is related to something you don't know about so you should ignore it. The last part, `[resolution 7,9]` describes how the clause was generated e.g. this was generated by resolution from clauses `7` and `9`.

This problem is not big enough yet to be able to explore different clause selection strategies but you can control how Vampire performs clause selection using the `--age_weight_ratio` option where you

specify a ratio between selecting clauses by age and by weight e.g. `10:1` says that for every 10 clauses selected by age you select one by weight.

On Equality. In Vampire proofs you will see steps marked as *superposition*. You should read this as *paramodulation*. Superposition is a lifting of parmodulation to use orderings in the same way that ordered resolution lifts resolution. However, it is a lot more complicated with equality so it is called something different. You may also see a rule called *demodulation*. This was mentioned in a lecture, it is also a special kind of paramodulation that works with unit equalities.

Missing Things

There are lots of things in Vampire that I haven't told you about. We want to turn off most of these during this lab so you should generally run Vampire using the following set of options

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

To help I have created a shell script `run_vampire` that wraps up Vampire and uses this set of options for you. If you don't use this then you might get different results in these and later exercises. You can just add extra options after this and they override the ones in the file e.g.

```
./run_vampire --selection 0 problem.p
```

If you're interested in what an option does then you can find out by running, for example,

```
./vampire -explain updr
```

Part 1: Understanding Proofs

This part is about being able to understand the proofs that Vampire produces.

Simple Problems

Consider these two simple problems are taken from Lecture 11.

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

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

For each problem you should do the following:

1. Translate the problem to TPTP
2. Clausify the problem by hand and compare this to the output of clausification with Vampire
3. Solve each problem using Vampire and record the proofs
4. For the first proof select a resolution step and identify the selected literals and the unifier required to perform the step
5. For the second proof select a paramodulation (superposition or demodulation) step and identify the term being rewritten and the unifier required to perform the step
6. Solve the first problem using unordered resolution (add `--selection 0`) and comment on how proof search changes. If you get a different proof then why?
7. Solve the second problem using `--selection 0 --age_weight_ratio 10:1` and comment on how proof search changes. Identify a paramodulation step in proof search that uses a non-empty substitution (i.e. it needs to bind some variables) and identify the term being rewritten and the unifier required to perform the step.

You should save your TPTP problems as `greed.p` and `dependency.p` respectively. Your exploration and answers to questions should be saved in a file `understand_proof_simple.txt`.

A More Complex Problem

Download the problem PUZ031+1.p from TPTP. And do the following:

1. Clausify the problem. How many clauses are there? How many Skolem functions are introduced?
2. Try and solve the problem using unordered resolution (add `--selection 0`). What happens?
3. Now run with the option `--statistics full` using the default given options. This tells you what Vampire did during proof search e.g. the number of clauses generated and activated and different kinds of inferences performed. Answer the following questions:
   (a) How many clauses were generated during proof search?
   (b) How many clauses were activated (selected)?

(c) How many times was the resolution rule applied?

(d) How long is the proof (i.e. how many steps)?

4. Play with the value of `age_weight_ratio` to see if selecting more clauses by age or weight generates fewer clauses overall for this particular problem

You should save your answers in `understand_proof_complex.txt`

Part 2: Understanding Saturation

So far we have just looked at proofs of entailment involving producing the empty clause. Here we briefly look at the converse; saturation showing consistency.
Consider this modified version of a previous problem

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

You should

1. Translate the problem into TPTP (this should just involve copying and modifying your previous file)
2. Try and prove the problem using default options and describe what happens and what this means
3. Repeat the previous step with unordered resolution (`--selection 0`) and explain why something different happens

You should save your answers in `understand_saturation.txt`.

Part 3: Modelling Something You Know II

In this part you should extend what you did in Part 3 of exercise 2A.

UPDATE: If you struggle to translate everything from your Prolog knowledge base into FOL then translate a non-trivial sub-part of it and explain what the problem was.

Step 1: To TPTP

Take the first-order formulas you produced in Step 3 of Part 3 of exercise 2A (these should be in a file `modelling_fol.txt`) and translate them into FOL. You should save your resulting file as `model.p`.

Hint 1. In TPTP one can write `fof(d,axiom,$distinct(a,b,c)).` as a shortcut for the formula `fof(d,axiom, a!=b & a!=c & b!=c)` e.g. that the constants are distinct. This only works on the formula level and only works with constants. It is useful for encoding unique names.

Hint 2. If you want to specify domain closure, e.g. that the only fruit are bananas or apples, then the standard way of doing this is

$$\forall x.(fruit(x) \rightarrow (x = apple \vee x = banana))$$

i.e. if something is a fruit then it is either an apple or banana.

Hint 3. If you have used lists in your Prolog knowledge base then you need to translate these into first-order logic. Lists can be captured by the function $cons(x, list)$ which constructs a list out of a new element x and another list $list$ and the constant empty list $empty$. You will also need to add some other axioms:

- $\forall x, list.cons(x, list) \neq empty$
- $\forall x_1, x_2, list_1, list_2.(cons(x_1, list_1) = cons(x_2, list_2) \rightarrow (x_1 = x_2 \wedge list_1 = list_2))$

To prevent infinite models of lists we also need to ensure that lists are acyclic, which requires an acyclic *sublist* relation:

- $\forall x, list.(sublist(list, cons(x, list)))$
- $\forall x, y, z.((sublist(x, y) \wedge sublist(y, z)) \rightarrow sublist(x, z))$
- $\forall x.\neg sublist(x, x)$

Note that, due to Warning 1 below, this is a conservative extension of the theory of lists as we cannot use first-order logic to make *sublist* the true transitive closure of the direct sublist relation.

Warning 1. We cannot represent the *transitive closure* of a relation in first-order logic. We've done this quite a bit in Prolog so that might seem strange. What we can do is state the a relation is transitive (the definition should be obvious) but to say that one relation is the transitive closure of another requires us to say that it is the *smallest* such relation and first-order logic isn't strong enough to say this as it requires us to quantify over relations (which we can do in second-order logic but not first-order). So why can we do it in Prolog? Because of the closed-world assumption which implicitly quantifies over all defined relations and says that they are the smallest relations satisfying the given constraints! What does this mean practically? You should translate your transitive relations as transitive relations and any entailments will still do what you expect but you cannot trust saturations as the associated model may not relate to the smallest relation. In other words provable things are the same but non-provable things are not.

Warning 2. If your model contained arithmetic then you can use the built-in arithmetic language in TPTP but remember from Lecture 12 that reasoning will become undecidable in general (and more difficult in practice). You may realise that you didn't need arithmetic but just some ordering. In which case, it is best to remove arithmetic from your model at this stage.

Step 2: Answering Simple Queries

Next translate some simple queries of your knowledge base to extensional form. Let us demonstrate with a little example. Imagine your Knowledge Base was

```
fof(one,axiom, vegetable(carrot)).
fof(two,axiom, vegetable(lettuce)).
fof(three,axiom, vegetable(cucumber)).
fof(four,axiom, fruit(apple)).
fof(five,axiom, fruit(banana)).
fof(distinct,axiom, ![X] : ~(fruit(X) & vegetable(X))).
fof(distinct,axiom, $distinct(carrot,lettuce,cucumber,apple,banana)).
fof(six,axiom, ![X] : (food(X) <=> (vegetable(X) | fruit(X)))).
```

i.e. that we have some vegetables and fruit, that something cannot be a fruit and a vegetable, that all named vegetables and fruit are distinct, and that all food is either vegetable or fruit. Note that I haven't specified that the only fruit/vegetables are the ones named (domain closure).

If we wanted to know if there is some food that is not a vegetable then we want to ask if such a thing exists e.g.

$$\exists x.(fruit(x) \wedge \neg vegetable(x))$$

Expressing this as a conjecture will produce a proof e.g.

```
4. fruit(apple) [input]
6. ! [X0] : ~(vegetable(X0) & fruit(X0)) [input]
8. ! [X0] : (food(X0) <=> (fruit(X0) | vegetable(X0))) [input]
9. ? [X0] : (~vegetable(X0) & food(X0)) [input]
10. ~? [X0] : (~vegetable(X0) & food(X0)) [negated conjecture 9]
12. ! [X0] : (~vegetable(X0) | ~fruit(X0)) [ennf transformation 6]
13. ! [X0] : (vegetable(X0) | ~food(X0)) [ennf transformation 10]
14. ! [X0] : ((food(X0) | (~fruit(X0) & ~vegetable(X0))) & ((fruit(X0) |
vegetable(X0)) | ~food(X0))) [nnf transformation 8]
15. ! [X0] : ((food(X0) | (~fruit(X0) & ~vegetable(X0))) & (fruit(X0) |
vegetable(X0) | ~food(X0))) [flattening 14]
29. fruit(apple) [cnf transformation 4]
31. ~fruit(X0) | ~vegetable(X0) [cnf transformation 12]
34. ~fruit(X0) | food(X0) [cnf transformation 15]
35. ~food(X0) | vegetable(X0) [cnf transformation 13]
36. ~vegetable(apple) [resolution 31,29]
41. food(apple) [resolution 34,29]
46. vegetable(apple) [resolution 41,35]
48. $false [resolution 46,36]
```

And from this we could infer that the food that is not a vegetable was an `apple` but doing this by looking at the proof is cumbersome.

Vampire has a *question answering* mode. If we run

```
./run_vampire kb.p --question_answering answer_literal
```

then we get

```
% SZS answers Tuple [[apple]|_] for kb
```

Can you work out how Vampire does this (the proof will look a bit different in this mode)?

In this step you should write three or more conjectures of the form

```
fof(query,conjecture, ?[X] : <formula involving X>).
```

that relate to simple queries of your Prolog knowledge base (you can existentially quantify over more than `X`). Save these (and only these, one per line) in `fol_queries.txt`. We will run them against your submitted `model.p` knowledge base.

Step 3: More Queries

Finally, translate the queries from Step 3 of Part 3 of lab exercise 2A into conjectures in first-order logic and append these to `fol_queries.txt`.

It is likely that these will not be extensional queries. For example, I might want to ask my above knowledge base if being a fruit implies being food e.g. the query

```
fof(query, conjecture, ![X] : (fruit(X) => food(X))).
```

Running Vampire should give a proof and we can conclude that the answer is 'yes'.

Submission and Marking Scheme

The submission site will open (covering lab 2A and 2B) by the end of 15th March and the marking scheme will be available then.