COMP24412
Academic Session: 2021-22

Lab 2: Reasoning in FOL

Francisco & Giles

A few points before we begin:

- Submission is via git for this lab - see the section at the end of this document about submission. You should use your `comp24412` repository (branch `lab2`). To get the provided scripts you will need to run

  ```
  git checkout lab2
  ./refresh.sh
  ```

- This lab isn't about writing code.
- Most of the lab is about running the Vampire theorem prover and explaining the results. If you fully understood what happened in lectures then exercises 1-5 should take less than 30 minutes.But I expect that this won't be the case for most people and it will be necessary to revise some concepts.
- The last part is about modelling something in FOL. You can be as ambitious as you want here but be warned - it can be addictive. I suggest (i) writing a simple fallback knowledge base to start with, (ii) keep lots of copies of your knowledge base as you try out different things - you will make a change you don't like and want to 'roll back'!
- Some of the activities don't have any associated marks - these are clearly labelled.
- In addition, every place where you can get marks is signposted with symbol ☞

Good luck and have fun!

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 (and related concepts e.g. ordered resolution, redundancy, reasoning with theory axioms etc) given in lectures
4. Use Vampire to answer queries on a 'real' knowledge base

Activity 1: Starting with Vampire (Unmarked)

We've looked at simple reasoning methods which could be implemented in a few 100 lines of Python. In this part we're going to play with a complicated reasoning tool. The Vampire theorem prover consists of about 200k of C++ code, although the parts you will use probably use less than half of this.
In this activity you'll get used to using Vampire. Nothing is submitted or assessed. The aim here is to run Vampire on a few small problems and in doing so learn

1. How to get hold of Vampire and where to look for more information
2. How to represent first-order formulas in the TPTP syntax
3. How to run Vampire form the command line
4. How to read Vampire proofs

Getting Vampire

Before you can begin you'll need to grab a version of Vampire. You can either download a pre-compiled binary or build your own (e.g. on your own machine) by following instructions here: `https://vprover.github.io/download.html`. Any relatively modern version of Vampire will work, although you'll need a version with Z3 for parts of Activity 7. There are additional notes on getting Vampire on Blackboard.

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
- Giles 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

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.

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

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)).
```

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

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 usually use `fof` standing for *first-order form*, `cnf` standing for *conjunctive normal form*, or `tff` standing for *typed first-order form*.
- `name` is a name given to the formula. Vampire only uses this in proofs. It doesn't impact proof search at all. Names don't even need to be distinct.
- `kind` describes the kind of formula. We will mainly use `axiom` and `conjecture` although later you will see us using `question` as well. 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

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 syntactically correct by running `./vampire --mode ouptut problem.p` where you put your TPTP problem in `problem.p`. This will print your problem back to you if you got the syntax right or give you a (hopefully helpful) error message if you did not.

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 should consist of two reasoning steps labelled `resolution` and `subsumption resolution`. Remember, this second rule is a special kind of resolution that we perform as a simplification. We don't see the difference in the proof as this just impacts the search space (we delete one of the parents of the resolution). So, for the proof you can think of `submsumption resolution` just as `resolution`. However, remember that the `submsumption resolution` rule is applied at a different point in proof search than `resolution`. We can turn this off with the option `-fsr off`.

Now try running

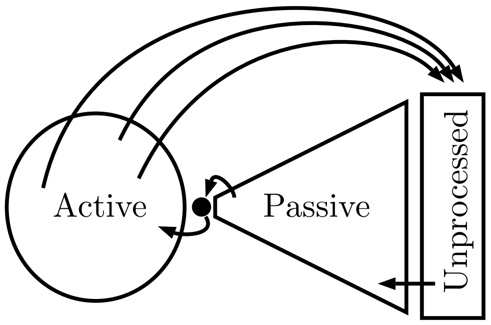
```
./vampire -fsr off --selection 0 rich.p
```

Do you get a different proof? You should because we have changed how we are doing literal selection for ordered resolution. 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.

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. I have also added the option `--proof_extra free` to include some extra information in the output. Without setting selection to 0 you should get the following new and active clauses

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

and with selection as 0 you should get

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

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) [resolution 7,9] {a:1,w:2,nSel:1,goal:1}
```

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 id number of the clause - clauses are numbered as they are produced. Then we have the clause `~rich(giles)` in cnf format. The next bit, `[resolution 7,9]` describes how the clause was generated e.g. this was generated by resolution from clauses `7` and `9`. Finally, the extra string `{a:1,w:2,nSel:1,goal:1}` gives the age of the clause, the weight of the clause, the number of selected literals in the clause (which are always the leftmost literals in active clauses), and whether this clause is derived from the goal. This is the information that is used for clause and literal selection.

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. For now you should add the option

```
--equality_proxy RSTC
```

to translate equality into a special predicate as described in lectures. If you drop this then you will see steps marked as *superposition* or *demodulation*, which are techniques mentioned briefly in the final part of lectures.

WARNING: Remember that equality here is different from the notion of equality we met in the Prolog fragment. There we were working with assumptions such as unique name assumption and a term algebra interpretation where equality was the same as syntactic equality (e.g. unifiability). Here equality is a binary predicate with some well-defined axioms.

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 -ep RSTC
```

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
```

You need to have the `vampire` binary in the same directory for the script to work.

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

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

Entailment and Question Answering

The above shows how to check ground entailments by writing a ground conjecture such as the conjecture `fof(three,conjecture, happy(giles))`. We can also use conjectures to ask non-ground conjectures e.g. we can ask if there exists somebody who is happy as `fof(three, conjecture, ?[X] : happy(X))` - what does the proof look like? We can also check universal entailments e.g.

```
fof(a,axiom, ![X,Y] : (f(X,Y) => ~f(Y,X))).
fof(a,conjecture, ![X] : ~f(X,X)).
```

which checks whether `f` being anti-symmetric entails `f` being irreflexive.

This is fine but we aren't getting the nice answers we had for Datalog and Prolog. As mentioned in lectures, we can use a trick to extract these. If we add `--question_answering answer_literal` to the Vampire options and run with an existential conjecture we get an answer tuple. This works by rewriting the conjecture such that solving the conjecture would involve a single final resolution step and then intercepting this step.

At the time of writing advanced features for question answering in Vampire exist in an experimental branch but I am exploring putting them into the master branch and this text will be updated if I do this. Currently Vampire will only return one answer but the advanced features allow it to return many answers. Note that in general, due to the open-world interpretation in first-order logic, there may no longer be a notion of *all of the answers*.

Exercise 2: Exploring Some Simple Proofs

Consider these two simple problems:

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

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

For each problem you should do the following:

☞ $\boxed{1}$ Translate the problem to TPTP

☞ $\boxed{2}$ Clausify the problem by hand and compare this to the output of clausification with Vampire

☞ $\boxed{3}$ Solve each problem using Vampire and record the proofs

☞ $\boxed{4}$ For the first problem:

(a) Solve the problem using the default options (the ones provided in the `run_vampire` script) and select a resolution step and identify the selected literals and the unifier required to perform the step

(b) 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?

☞ $\boxed{5}$ For the second problem:

(a) Solve the problem using default options (including `-ep RSTC`)

(b) Solve it again adding `--selection 0 --age_weight_ratio 10:1` and comment on how proof search changes.

(c) Now add `-ep off` (keeping the options from the previous step) and comment on what happens when we don't translate away equality. Can you identify a non-trivial case (e.g. where some unification is required).

You should save (and submit) your TPTP problems as `greed.p` and `dependency.p` respectively. Your exploration and answers to questions should be saved in a file `exercise2.txt`.

Exercise 3: Exploring a More Complex Proof

Download the problem PUZ031+1.p from TPTP. Back in the 80s this was a problem that was famously too difficult to solve due to the state space explosion.

Do the following steps and answer the associated questions:

☞ $\boxed{1}$ Clausify the problem. How many clauses are there? How many Skolem functions are introduced?

☞ $\boxed{2}$ Try and solve the problem using unordered resolution (add `--selection 0`). What happens?

☞ $\boxed{3}$ 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)? (*Hint*: this is not the same as the number of the last clause as many generated clauses are not included in the proof)

☞ $\boxed{4}$ 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. Remember, `-awr 10:1` selects 10 clauses by age for every 1 it selects by weight and vice versa for `-awr 1:10`.

☞ $\boxed{5}$ Run with `--show_reductions on` and extract and explain a subsumption. Recheck the statistics and see how many subsumption deletions there were.

☞ $\boxed{6}$ Now let's turn subsumption deletion off with `--forward_subsumption off`. How does this impact (i) the number of generate clauses, and (ii) the length of the proof?

You should save your answers in `exercise3.txt`

Exercise 4: Exploring Saturations

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

☞ $\boxed{1}$ Translate the problem into TPTP (this should just involve copying and modifying your previous file)

☞ $\boxed{2}$ Try and prove the problem using default options and describe what happens and what this means

☞ $\boxed{3}$ Repeat the previous step with unordered resolution (`--selection 0`) and explain why something different happens

☞ $\boxed{4}$ Now run with the option `--unused_predicate_definition_removal on` and explain why something different happens. It might help to run `./vampire -explain updr`.

☞ $\boxed{5}$ Finally, run with the option `-sos on` and explain why something different happens. Again, running `./vampire -explain sos` is likely to help. It might also help to refer back to the above description of the given clause algorithm and consider what saturation means.

You should save your answers in `exercise4.txt`.

Activity 5: Exploring Model Building (Unmarked)

I haven't spoken to you about model building but it can be interesting to see alternative reasoning methods. Take the problem from Exercise 4 (on saturation e.g. we know that the problem is consistent) and add an axiom to express $giles \neq money$ and then run

```
./run_vampire --saturation_algorithm fmb problem.p
```

What you will get is a *finite model* that makes the axioms true but the conjecture false. This is why the reported status is `CounterSatisfiable`. The finite model building process implemented in Vampire works by translating the problem of finding a model of a particular size into a SAT problem and iterating this process for larger model sizes.

Exercise 6: Modelling and Answering Questions

This is a more open-ended exercise. The work required to get full marks is high (but it shouldn't take that much effort to get most of the way there). There are four steps for you to follow - you should pick a domain, model it, check your model makes sense, and then write some conjectures representing queries that you want to check for entailment. See below for details.

Your knowledge base in TPTP format should go in `exercise6.p` and your conjectures should go in `exercise6_conjectures.p` with one conjecture per line in TPTP format (so we can check them automatically - you don't need to put any explanations in this file). Finally, you should write a brief explanation of what you've done in `exercise6.txt` - we suggest including a heading per step and writing something beneath each heading. In particular, if you find that you need to run Vampire in a certain way to get things to work then mention this in `exercise6.txt`.

☞ **Step 1 - The Domain.** You can pick anything you want but you will probably find it easier if you pick something where there are some clear extensional relations (e.g. objects you can describe the properties of) and some clear intensional relations (i.e. those defined in terms of other relations). Ideally you will have a recursively defined relation. Some ideas include modelling a football league, different pizza toppings, parts of the body, the rules of a simple game, or your degree courses.

☞ **Step 2 - Modelling.** Use the TPTP language to model your chosen domain. By now you should be familiar with the TPTP syntax (see above for links if not). Remember that parenthesis are important for defining the scope of quantification! You will need to choose either the `fof` or `tff` language. In general, you are likely to have the following things in your model:

1. If you choose `tff` then you will need to define the signature e.g. the types of the symbols you will use.
2. Defining some individuals (constants) and their properties and relations to each other (using ground facts). You will probably want to declare that these individuals are unique and you may want to *close* a particular (sub)domain - see below for hints.
3. Defining some helper predicates (relations). For example, you may want to define a structure (e.g. lists), an ordering, or a way of converting between objects.
4. Defining the main predicates (relations) and their relation with each other. You *should* aim to make use of first-order features that are not available in the Datalog or Prolog fragments e.g. disjunctive 'rule heads' and (nested) existential quantification.

In the last section of this document Giles gives more detailed hints on modelling and a couple of examples.

☞ **Step 3 - Checking your Knowledge Base.** Ideally your knowledge base will be consistent and minimal. Write down some thoughts in `exercise6.txt` on how you can do this with Vampire. Why do we want our knowledge base to be consistent? What might we mean by minimal? As an example, consider this small knowledge base:

$$\begin{array}{l}
\forall x.\mathit{playsIn}(x, \mathit{homeGround}(x)) \\
\forall x.(\neg \mathit{team}(x) \vee \mathit{hasManager}(x, \mathit{managerOf}(x))) \\
\forall x.(\neg \mathit{team}(x) \vee \mathit{womansTeam}(x) \vee \mathit{hasManager}(x, \mathit{managerOf}(x))) \\
\mathit{homeGround}(\mathit{arsenal}) = \mathit{emirates} \\
\mathit{playsIn}(\mathit{arsenal}, \mathit{emirates})
\end{array}$$

Are there any axioms that could be removed without changing what is true?

☞ **Step 4 - Queries and Conjectures.** Finally, you should write some queries (e.g. ground or existential conjectures) and/or conjectures that you want to check on your knowledge base. You can ask things that you expect to be entailed and those you don't expect to be but in the latter case you won't get an answer if you use arithmetic (recall that reasoning in FOL with arithmetic is always incomplete). If you can't get Vampire to answer your queries/conjectures in the default mode provided try the portfolio mode which uses a lot of options you haven't met in a long sequence of strategies. (The next activity invites you to experiment with `--mode portfolio`)

Activity 7: Exploring Arithmetic (Unmarked)

Let's take a look at how Vampire deals with arithmetic, a common domain required for modelling interesting things. In lectures we heard that reasoning with arithmetic is hard. Let's look at some examples.

Download the following four problems:

- GEG025=1.p - Given some information about cities show that there exists a big nearby city.
- ARI744=1.p - Showing that the definition of square root entails the square root of 4 being 2.
- MSC023=2.p - Demonstrating Fahrenheit 451 to Celcius conversion
- DAT105=1.p - Showing that a certain definition of lists entails a given property

Write out the conjectures in English. You may want to break it down into bullet points, especially for the first one (you can summarise what the two `d` and `inh` predicates are doing).

Now try and solve each problem with Vampire. Here you should use Vampire directly instead of via the `run_vampire` script as the problems are hard and you want to use the things that this script turns off - in particular equality proxy doesn't work with arithmetic. By default Vampire will add *theory axioms* and try and use them to solve the problem. Which problems can Vampire solve in this way? Which theory axioms did we use (look at the proof)?

There should be at least one problem that we cannot solve (in any reasonable time) just using theory axioms. There are two approaches that can help in such circumstances:

- Running with `-sas z3` runs the method mentioned in lectures that uses an SMT solver, as described in my paper *AVATAR Modulo Theories*. This helps if some ground part of the search space is theory inconsistent.
- Running with `-thi all -uwa all` is the most general way of running the techniques described in Giles' paper *Unification with Abstraction and Theory Instantiation*. Together these help make unification aware of theories e.g. to allow us to resolve things when we can make things equal in the theory, not just syntactically.

Do either of these help with the problem(s) you couldn't solve? Can you see which bit of the proof uses the new options (don't worry if you can't).

Finally, there is the *'killer'* option of `--mode portfolio` that runs Vampire in a mode where it attempts lots of different complicated strategies in sequence. This kind of portfolio mode (there are others) is how Vampire is typically run in the 'real world'. If you still haven't solved one of the above problems, does this help? If so, what is the 'strategy' string that is used?

A strategy string is something like

```
dis+1011_10_add=large:afr=on:afp=4000:afq=1.0:amm=off:anc=none:lma=on:nm=64:nwc=4:sac=on:sp=occurrence_2
```

and is an encoding of the set of options that define a strategy. You can ask Vampire what any of the named options mean e.g. `./vampire -explain nm` but the explanations for some of the more experimental options can be a bit vague!

Submission and Mark Scheme

As usual, submission is via `git`. You must work on the `lab2` branch and *tag* your submission as `lab2_solution`. The provided `submit.sh` script does this for you. Note that you must push your tagged commit to Gitlab before the deadline (it is not good enough just to make the commit locally).

Your work will be marked offline and returned to you with comments.

The marking scheme is given below. Note that the marks may seem a bit uneven. I have purposefully given Exercise 2 more weight as it contains the fundamental things I want you to take away from this lab. The challenging parts are the second marks of exercise 4 and the final two marks of exercise 6 (e.g. the difference between 80% and 100%).

In general, written answers do not need to be long (some answers may be a few words) but they need to address the question being asked.

Mark Scheme:

Ex 2 (6 marks) - Three marks for each problem (1 mark for steps 1+2, 2 marks for steps 2+4/5). Full marks for good answers, half marks for reasonable attempts.

Ex 3 (4 marks) - There are 0.5 marks for each question except questions 3 and 4 which are worth 1 mark each.

Ex 4 (2 marks) - There are 0.5 marks each for steps 2-5.

Ex 6 (6 marks) - This is split as follows:

- 4 marks for the knowledge base itself. Something that is simple but (i) goes beyond what we can do in Prolog/Datalog and (ii) works (e.g. the Simple Example) gets 2 marks. An extra mark for additional complexity (e.g. good use of functions, recursive relations, and/or quantifier alternation). The final mark is for something impressive (you can ask). **A maximum of 2 marks will be awarded if the explanation is inadequate.**
- 1 mark for the set of conjectures. For full marks you need at least four conjectures, with at least one being universally quantified, and a discussion for each conjecture covering whether it holds.
- 1 mark for the answer to Step 3 (0.5 for thoughts on consistency, 0.5 for thoughts on minimality)

Hints for Exercise 6

This section can be updated on request although also see the forum.

You should apply Occam's razor - simplicity is best for two reasons: (1) it's easier to write, and (2) it will be easier to do reasoning. These two are often at odds. For example, for (2) it can be better to avoid equality and the built-in arithmetic language but these can help with (1) e.g. make things easier to write. Don't be afraid to reduce, reformulate or remove something later if it turns out to be unhelpful for reasoning.

Some further hints:

- 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.
- 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.
- In the `tff` language you can introduce the notion of *sorts* to reduce the amount of guarding you need to do in different places. For example, if I have sorts I can replace the above domain closure axiom with a non-guarded one:

  ```
  tff(fruitType, type, fruit : $tType).
  tff(appleType, type, apple : fruit).
  tff(bananaType, type, banana : fruit).
  tff(dcf, axiom, ![X:fruit] : (X=apple | X=banana)).
  ```

  although in this case the guard $\neg fruit(x)$ is something that can be selected, thus blocking unnecessary use of the axioms in certain cases - so you might make reasoning harder by removing it. If that doesn't make sense, try the two alternatives and see what happens!
- If you're tempted to use numbers and orderings on them just because you want to order things then pause and remember to keep things simple. Numbers come with lots of extra baggage. You probably just want to add your own ordering with axioms. Consider whether you really need transitivity as in general it slows down proof search. If you have a small (and finite) number of objects you are ordering you might find it easier to make the ordering explicit.
- Remember that equality in FOL is not the same as equality in Prolog where equality was just syntactic equality. This is helpful, as it allows us to write useful things, but it is also important that you don't rely on this.
- TPTP doesn't have a native idea of lists. 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:

– $\forall x, list.(sublist(list, cons(x, list)))$
– $\forall x, y, z.((sublist(x, y) \land sublist(y, z)) \to sublist(x, z))$
– $\forall x.\neg sublist(x, x)$

Note that, due to the remark 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.

Remark. We cannot represent the *transitive closure* of a relation in first-order logic. We do it all the time 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? Any proof using a transitive relation still holds for the smallest such relation, but non-proofs (e.g. saturations) may not as we don't know which relation is used by the associated model.

A Simple Example. Imagine your domain is about fruit and vegetables. You might write the following simple knowledge base which describes some vegetables and fruit.

```
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).

I might have initially added

```
fof(tomato, axiom, vegetable(tomato)).
fof(tomato, axiom, fruit(tomato)).
```

but then found that my knowledge base was consistent. I could try and add some exceptions to my rules e.g. I could extend my knowledge base as follows (keeping axioms 1-6 the same):

```
fof(tomato, axiom, vegetable(tomato)).
fof(tomato, axiom, fruit(tomato)).
fof(exception,axiom, ![X] : (exception(X) <=> (X=tomato))).

fof(distinct,axiom, ![X] : (~exception(X) => ~(fruit(X) & vegetable(X)))).
fof(distinct,axiom, $distinct(carrot,lettuce,cucumber,apple,banana,tomato)).
```

Now I have one place I can list my exceptions to the 'fruit and vegetables are different' rule and I use this exception to guard the mutual exclusion rule.

If we wanted to know if there is some food that is not a vegetable (it's not guaranteed just by axiom six) then we want to ask if such a thing exists e.g.

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

Expressing this as a conjecture

```
fof(query,conjecture, ?[X] : ( food(X) & ~vegetable(X))).
```

will produce a proof with some final steps looking something like this

```
46. vegetable(apple) [resolution 41,35]
48. $false [resolution 46,36]
```

That is, before we added exceptions. After adding them we need to show that we are not in an exceptional case so the last step will probably involve `apple != tomato`. 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. We can use the *question answering* mode mentioned earlier. If we run

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

then we get

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

Next I might want to know whether being a fruit implies being a food. I could check this by posing the conjecture

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

which is something we can't do in Prolog. Here we are looking for a `Theorem` result - it's not an existential query so we don't use question answering. If Vampire returns `Theorem` it means that the conjecture is always true (valid), thus the entailment holds.

If I posed a different conjecture e.g.

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

I might get a `CounterSatisfiable` answer, which means that there is a model for the knowledge base that is also a model for the negated conjecture e.g. the conjecture does not hold in all models of the knowledge base and is not entailed by it.

If you use the experimental question answering branch (see Blackboard) you can get more answers e.g.

```
./vampire_qa  --question_answering answer_literal -av off fruit.p
% SZS answers Tuple [[banana],[apple]|_] for fruit
```

This experimental branch won't terminate when it finds one answer but carry on going until there are no more answers or it has found `question_count` answers (specified by an option). Because of this, you won't get a proof (as there would need to be a proof per answer).

An Advanced Example. Most of you will have a knowledge base that models some real world thing with individuals, properties, and relations. That's fine and you can (quite straightforwardly) get full marks for a knowledge base of that kind. But you can also model something a bit more abstract. Here I have a go at modelling the quicksort algorithm and attempt to show that it always sorts lists. I don't quite get to the end of this journey but the journey would definitely count as 'impressive' in the mark scheme.

The first thing I need to do is define our notion of lists and ordering, both necessary for the question of sorting. As per the above hints, I can do this as follows. Note that I introduce my own ordering rather than relying on something like < from the theory of arithmetic.

```
% Basic List axioms
fof(l1, axiom, ![X,L] : cons(X,L) != empty).
fof(l2, axiom, ![X1,X2,L1,L2] : ( cons(X1,L1) = cons(X2,L2) => (X1=X2 & L1=L2))).
fof(l3, axiom, ![X,L] : sublist(L,cons(X,L))).
fof(l4, axiom, ![X,Y,Z] : ( ( sublist(X,Y) & sublist(Y,Z)) => sublist(X,Z) )).
fof(l5, axiom, ![X] : ~sublist(X,X)).

% ordering of elements
fof(o1, axiom, ![X,Y] : ( (ord(X,Y) & ord(Y,X)) => X=Y)). % antisymmetry
fof(o2, axiom, ![X,Y,Z] : ((ord(X,Y) & ord(Y,Z)) => ord(X,Z))). % transitivity
fof(o3, axiom, ![X] : ord(X,X)). % reflexivity
fof(o4, axiom, ![X,Y] : (ord(X,Y) | ord(Y,X) )). % totality
```

Note that axioms `l4` and `o2` are dangerous. They allow me to generate arbitrarily complex consequences. One way of taming either (a little) would be to stratify by using a 'direct' sublist/less-than relation. These relations are the main (but not only) reason that some reasoning steps below don't work. In some cases I could turn off (comment out) some of these axioms as they are not always useful for proof search.

The next thing I'm going to specify is sortedness. I introduce a predicate on lists that is meant to be true if that list is sorted (wrt my ordering relation).

```
% sorted
fof(s1, axiom, sorted(empty)).
fof(s2, axiom, ![X] : sorted(cons(X,empty))).
fof(s3, axiom, ![X,Y,L] : (sorted(cons(X,cons(Y,L))) <=> (sorted(cons(Y,L)) & ord(X,Y)))).
```

At this point I can test this using a few conjectures. For this I need some ordered elements. I can either add some extra facts to my knowledge base for testing i.e.

```
% My elements
fof(e1, axiom, ord(a,b) & ord(b,c) & ord(c,d) & ord(d,e) & ord(e,f) & ord(f,g)).
fof(e2, axiom, $distinct(a,b,c,d,e,f,g)).
```

and use these to ask conjectures such as

```
fof(c, conjecture, sorted(cons(a,cons(b,cons(c,cons(d,empty)))))).
```

or I could include the ordering information in the conjecture e.g.

```
fof(c,conjecture, ($distinct(a,b,c) & ord(a,b) & ord(b,c))
                                        => sorted(cons(a,cons(b,cons(c,empty))))).
```

Now I can start so specify quicksort but before I do that I realise that I'm going to need a function to append lists. So I specify this as

```
% append
fof(a1, axiom, ![L]: append(empty,L) = L).
fof(a2, axiom, ![X,L1,L2]: append(cons(X,L1),L2) = append(L1,cons(X,L2))).
```

and can test it using conjectures such as

```
fof(c, conjecture, append(cons(a,empty),cons(b,empty)) = cons(a,cons(b,empty))).
```

At this point I could try and establish more general properties such as the length of *append*(X, Y) is equal to the length of X plus the length of Y. However, I would need induction for this and first-order logic isn't strong enough for that by itself - we'll return to this point later.

Okay, now I'm ready to specify quicksort and I do so as follows:

```
% quicksort
fof(q1, axiom, quicksort(empty) = empty).
fof(q2, axiom, ![L] : (L!=empty => quicksort(L) =
                        append(quicksort(left(L)),cons(pivot(L),quicksort(right(L)))))).
```

I made some choices here that could have been made differently. I specify quicksort simply in terms of a recursive call to a left and right partition of the list. The left and right partitions are defined in terms of a general partition function that I define next. At this point I reflect that FOL is not a programming language but my definitions here look quite similar to something I might write in a logical or functional programming language.

For partition I choose to separate out the different cases rather than attempt to put them all in one definition. But I have decided to have a single partition predicate that takes an 'order' argument rather than defining something like `filterLess` and `filterGreater`. Note that `partition` and `left` and `right` are undefined for empty lists. This doesn't mean they don't equal something, I just don't constrain what they are allowed to equal.

```
% partition
fof(p1, axiom, ![L] : (L!=empty => left(L)  = partition(L,pivot(L),less))).
fof(p2, axiom, ![L] : (L!=empty => right(L) = partition(L,pivot(L),greater))).
fof(p3, axiom, ![E,C] : partition(empty,E,C) = empty).
fof(p4, axiom, ![E,X,L] : ((ord(X,E) & X!=E) =>
                        partition(cons(X,L),E,less) = cons(X,partition(L,E,less))))).
fof(p5, axiom, ![E,X,L] : ((ord(E,X))
                        =>  partition(cons(X,L),E,less) = partition(L,E,less))).
fof(p5, axiom, ![E,X,L] : ((ord(E,X) & X!=E)
                        =>  partition(cons(X,L),E,greater) = cons(X,partition(L,E,greater)))).
fof(p6, axiom, ![E,X,L] : ((ord(X,E))
                        =>  partition(cons(X,L),E,greater) = partition(L,E,greater))).

% pivot picks the first element. I could try something more complicated but that
% would probably be too complicated
fof(p7, axiom, ![X,L] : pivot(cons(X,L)) = X).
```

It might have been more general to write `partition` as

```
fof(p8, axiom, ![E,X,L,C] : ((((C=greater & ord(E,X) & X!=E)) | (C=less & ord(X,E) & X!=E))
                        =>  partition(cons(X,L),E,C) = cons(X,partition(L,E,C)))).
fof(p9, axiom, ![E,X,L,C] : ((((C=greater & ord(X,E))) | (C=less & ord(E,X)))
                        =>  partition(cons(X,L),E,C) = partition(L,E,C))).
```

But that gets a bit complicated and the extra structure can bet in the way during proof search.

I test the partition function with a few of conjectures:

```
fof(t1,axiom, test  = cons(b,cons(g,cons(c,cons(d,cons(a,cons(f,empty)))))) ).
fof(c, conjecture, partition(test,c,less) = cons(b,cons(a,empty))).
fof(c, conjecture, partition(test,c,greater) = cons(g,cons(d,cons(f,empty)))).
```

Remember that we only add one conjecture at a time. But trying to solve them with default mode times out. So I resort too `--mode portfolio` and one of the strategies solves each problem (I have two conjectures so two problems). I've minimised the strategy to

```
ott+1002_8:1_bd=off:bs=unit_only:fsr=off:nm=32:nwc=10:sp=occurrence:urr=on:updr=off_14
```

based on my understanding of what the options mean. A few options here are interesting. The `nwc` option makes proof search more goal-directed. The `sp` option changes the symbol ordering. The options `bd`, `bs`, `fs`, and `updr` are turning off certain simplifications.

Now that I'm finished I can test the quicksort function with a number of ground queries

```
fof(c, conjecture, quicksort(cons(a,cons(b,empty))) = cons(a,cons(b,empty))).
fof(c, conjecture, quicksort(cons(b,cons(a,empty))) = cons(a,cons(b,empty))).
fof(c, conjecture, quicksort(cons(c,cons(b,cons(a,empty)))) = cons(a,cons(b,cons(c,empty)))).
fof(c, conjecture, sorted(quicksort(cons(a,cons(b,empty))))).
fof(c, conjecture, sorted(quicksort(test))).
```

Again, I need portfolio mode to solve these. The last one requires me to turn off transitivity of `ord` in a bit of a hack as otherwise this transitivity swamps the search space.

At one point things weren't working and I used question answering mode to find out what was going wrong by asking

```
fof(c,conjecture, ?[X,L] : quicksort(cons(a,cons(b,empty))) = cons(X,L)).
```

Note that I cannot simply ask `?[L] : quicksort(cons(a,cons(b,empty))) = L` as in FOL we immediately have a term that equals `L` - it's `quicksort(cons(a,cons(b,empty)))`.

Finally, I can ask some more general questions about arbitary lists as follows.

```
fof(c, conjecture, ![X] : quicksort(cons(X,empty)) = cons(X,empty) ).
fof(c, conjecture, ![X] : (ord(X,Y)
                       => quicksort(cons(X,cons(Y,empty))) = cons(X,cons(Y,empty)) )).
fof(c, conjecture, ![X,Y,L] : ((ord(X,Y) & sorted(cons(Y,L)))
                       => quicksort(cons(X,cons(Y,L))) = cons(X,cons(Y,L)) )).
fof(c, conjecture, ![L] : sorted(quicksort(L))).
```

But currently Vampire can only establish the first one. The middle two shouldn't be out of its reach and I haven't explored what's going on (perhaps my model is a bit off). I know that the last one isn't possible without explicitly providing induction axioms (or using the in-built induction for the theory of datatypes but that requires rewriting the problem in a different format).