COMP24412
Lab 1: Reasoning in FOL
Lab session activities

Francisco Lobo and Joseph Razavi*
Academic session: 2023-24

Learning objectives

At the end of this laboratory exercise you should be able to:

1. Model problems using first-order logic (FOL).
2. Translate first-order formulas into the TPTP format.
3. Run Vampire on TPTP problems and explain its output; extracting proofs and models.
4. Relate Vampire options to the underlying refinements of the given-clause algorithm.

The first two relate to material you've been practising since the start of the course, the others to material on FOL reasoning that we'll develop from Week 4. There will be 2 lab sessions for this reasoning lab and in the first one you should complete the lab activities that follow; they introduce content to get you ready to work on the subsequent lab exercises.

Preliminaries

In this lab exercise we will use the Vampire theorem prover. This is installed on the Ubuntu image of the Kilburn lab machines, and you should be able to `ssh` into one of these to run Vampire. In case you'd like to get Vampire installed on your own machine, you can build it from source by following instructions at `https://github.com/vprover/vampire`. There you'll also find some pre-compiled Linux binaries available for download, and if you are on MacOS you can install Vampire using MacPorts. Any relatively modern version of Vampire will work, but note that you'll need a version with Z3 support to explore proofs that involve arithmetic.

Important: We only support the version of Vampire on Kilburn machines, so you must ensure that all your code works in the lab machines before submission!

Hopefully this lab manual is reasonably self contained, but if you want more information about Vampire here are some source to look into:

- Giles Reger gave a day tutorial on Vampire in 2017 and the material can be found on GitHub. The theory slides give a good introduction to the given-clause algorithm that Vampire uses.
- The unofficial technical manual for Vampire is the CAV 2013 tutorial paper. It gives a good theoretical overview of Vampire, how it works, and some of its options.

There are lots of Vampire features that we won't explore in this lab and we need to turn off most of these. For this reason, on the Kilburn lab machines the Vampire binary

```
$ vampire_z3_rel --version
```

was installed together with a wrapper shell script

```
$ run_vampire --version
```

that sets the correct default options to complete the lab exercises.

*The original version of this lab was developed by Giles Reger.

If you have Vampire on your own machine, then you can mimic the wrapper by setting a shell alias

```
$ alias run_vampire="vampire_z3_rel \
  -ep RSTC -updr off -fde none -nm 0 -s 1 -sa discount -fsr off -av off"
```

In order to explore theorem proving with Vampire, you'll need to re-enable some of these features and/or change additional Vampire options as indicated below; to do this you just need to give extra command-line arguments overriding those in the helper script (or alias) as necessary.

Important: You should always execute Vampire via the `run_vampire` wrapper, if you don't use it then you might get results that differ from this manual. In any case, for the exercises in this lab you'll need to keep track of which command-line arguments you use on your Vampire runs.

You can access extensive information about Vampire from the command-line

```
$ run_vampire --show_options on
```

Reading this documentation will help you understand the behaviour of Vampire. If you're interested in what a specific option does then you can run, for example

```
$ run_vampire -explain ep
```

In this case, Vampire reports that the option `ep` controls the proxy predicate that it can use to treat *equality*. You have seen how to introduce an equality proxy in the notes. If you turn it off, then you'll find that Vampire use advanced features like *superposition* or *demodulation*. You will see these in action in the exercises (even if they aren't part of the examinable content of the course).

Important: In first-order logic there is no assumption of unique names, and so equality isn't the same as syntactic "unifiability". Instead, equality in FOL is understood as a binary predicate with some well-defined axioms, which restricts the possible models for the particular theory.

Modelling problems with TPTP

You should aim to complete the activities in this part by the end of your first lab session. The goal is to practise the first two learning objectives on a toy example.

Using FOL to describe knowledge about a particular situation is one of the core objectives of this course unit. As an example, suppose that in some world:

(a) We define being happy as loving someone.

(b) We assume rich people love money.

(c) We know that Giles is rich.

It is clear that both *happy* and *rich* should be (unary) predicates about people. We have information about a specific person, which we can denote by a constant, say *giles*. Intuitively, we distinguish loving another person from the love of money. But in the ignorance of this semantic distinction, we could use a single binary predicate *loves* to syntactically encode these two kinds of love. We then represent the above knowledge in FOL as

(a) $\forall x.\, (\mathrm{happy}(x) \leftrightarrow \exists y.\, (\mathrm{loves}(x, y)))$

(b) $\forall x.\, (\mathrm{rich}(x) \rightarrow \mathrm{loves}(x, \mathrm{money}))$

(c) $\mathrm{rich}(\mathrm{giles})$

Given this knowledge we could ask if Giles is happy in this world. Mathematically, this is the same as checking if the axioms (a) to (c) entail the following conjecture

(d) $\mathrm{happy}(\mathrm{giles})$

To solve this problem using an automated theorem prover like Vampire, the first step is to write it down in a format that can be processed. We'll use the syntax of The TPTP Problem Library, which is almost exactly an ASCII version of the mathematical notation[1]. To learn about this library you can read the TPTP manual. You should start by working through the slides of the TPTP World Tutorial, in particular the examples for propositional logic and for first-order logic.

In the TPTP language the above problem can be written as follows.

```
fof(a, definition, ![X]:( happy(X) <=> (?[Y]: loves(X,Y)) )).
fof(b, axiom, ![X]:( rich(X) => loves(X,money) )).
fof(c, axiom, rich(giles)).
fof(d, conjecture, happy(giles)).
```

You should save this on a file called `greed.p` to use in the next activity. This knowledge-base consists of four declarations, each of which is terminated by a full stop. All of these declarations have the structure *form*`(`*name*`,`*role*`,`*formula*`).`

The ***form*** describes the logic being used.

Here we will usually use **`fof`** standing for *first-order form*, **`cnf`** standing for *clausal normal form*. You will also encounter **`tff`** standing for *typed first-order form*.

The ***name*** is an identifier given to the formula.

Vampire will only use this in proofs with the option `--output_axiom_names on`. These identifiers do not impact proof search at all! Names don't even need to be distinct.

The ***role*** describes the kind of formula.

We will mainly use **`definition`**, **`axiom`** and **`conjecture`**, but you will also see others like **`type`** being used. The meaning of a TPTP file is that the conjunction of the definition and axiom formulas entails the conjecture formula. There can be at most one conjecture. It is possible to have no conjecture, in which case we are just checking the consistency of the knowledge-base.

The ***formula*** is the logical formula being described.

The syntax is what one might expect for an ASCII based representation of FOL formulas. It is important to note that in TPTP formulas

- words that begin with upper-case letters are *variables*
- words that begin with lower-case letters are *constant*, *function*, or *predicate* symbols

You have seen the TPTP notation is the notes, but briefly the relevant logical syntax is

- `~` for not
- `&` for conjunction (and), `|` for disjunction (or)
- `=>` for implication, `<=>` for equivalence
- `![X]:` for universal quantification over `X`
- `?[X]:` for existential quantification over `X`
- `=` for equality and `!=` for disequality

Note that you can group quantifiers of the same kind. Thus `![X,Y]:` can replace `![X]: ![Y]:`

To practise your understanding of TPTP, try writing out the following formula in this syntax

$$\forall x.\,(P(f(x)) \rightarrow (\exists y.\,Q(x,y)))$$

If that was tricky, go back to and read this section again or ask for help. Now put your TPTP code in a file called `syntax.p` and then check whether it is syntactically correct by running

```
$ run_vampire --mode output syntax.p
```

1. Vampire accepts TPTP and also the SMT-LIB format, which is a lisp-based syntax used by SMT solvers.

This will print TPTP your problem back to you if you got the syntax right, or give you a (hopefully helpful) error message if you didn't.

The first thing that Vampire does when solving a problem is to put it into clausal form. You have studied how to do this by hand in the lectures, but you can make Vampire do it for you by running the following command.

```
$ run_vampire --mode clausify syntax.p
% Running in auto input_syntax mode. Trying TPTP
cnf(u5,axiom,
    q(X0,sK0(X0)) | ~p(f(X0))).
```

As you can see, the output of Vampire is itself in TPTP format. Does the result make sense to you? Can you write out the resulting clause in the first-order logic syntax we are used to? Now do the same with the `greed.p` file.

Important: When doing the lab exercises, it's a good idea to always check the clausal normal form that Vampire generates. This is an easy way to, for example, verify that you haven't missed any set of parenthesis.

Proving things with Vampire

You should aim to complete the activities in this part by the end of your assigned lab session. The goal is to practise the third learning objective on a toy example.

The `greed.p` problem we introduced in the previous section ask us to establish whether

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

The technique that Vampire uses for this is called *proof by refutation*, that is based on the result of Proposition 2 in the notes. Instead of asking if the set of premises (on the left) entails the conjecture (on the right), we negate the conjecture and try to show that the resulting set of formulas

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

is *unsatisfiable*, which means that it has no models. It turns out that this is something that can be done proof theoretically, because from a set of unsatisfiable formulas we must be able to derive `false`. It follows that to construct a proof simply we need to to work out all the consequences of a given set of formulas, with respect to a (complete) inference system for FOL, and check that one of these is `false`. The process of extending a set of formulas[2] with its consequences is known as *saturation*, and when doing proof by refutation we can stop this process as soon as we find `false`.

The main derivation rule of Vampire's inference system is called *resolution*. This is a generalisation of *modus ponens*, which you have studied for propositional logic in COMP11120. We'll be covering *unification* of FOL formulas in the lectures, which is a necessary technique to use modus ponens in first-order logic. As an example, given formulas (b) and (c) in the `greed.p` problem we can derive

$$\frac{\mathrm{rich}(x) \rightarrow \mathrm{loves}(x,\mathrm{money}) \quad \mathrm{rich}(\mathrm{giles})}{\mathrm{loves}(\mathrm{giles},\mathrm{money})}\ \big[x \mapsto \mathrm{giles}\big]$$

because $\big[x \mapsto \mathrm{giles}\big]$ is the *most general unifier (mgu)* for the formulas $\mathrm{rich}(x)$ and $\mathrm{rich}(\mathrm{giles})$. Under this substitution, the RHS of the implication is the derived formula $\mathrm{loves}(\mathrm{giles},\mathrm{money})$.

---

2. Conceptually the negated conjecture is no different from any of the other formulas in the set, but theorem provers like Vampire will treat conjectures differently so the proof search is goal oriented.

The formal definition of the resolution rule is

$$\frac{\neg A \vee F \quad B \vee G}{(F \vee G)\theta}\ \theta = \mathsf{mgu}(A, B)$$

where A, B are atoms and F, G arbitrary clauses. Note that when doing resolution we always treat clauses as sets of literals! Thus it is possible for F or G to be *empty clauses* in the above rule. For example, if $G = \{\}$ is the empty set of literals then the premise on the RHS simplifies because $\{B\} \cup G = \{B\}$. In FOL syntax this could be written as $B \vee \mathsf{false} \equiv B$.

Look again at the above modus ponens example using `greed.p` formulas, and convince yourself that it is an application of the formal resolution rule. You should find that in this case $A = \mathsf{rich}(x)$, $B = \mathsf{rich}(\mathsf{giles})$, $F = \mathsf{loves}(x, \mathsf{money})$, and G is empty. Once you gain experience with the resolution calculus, you'll get comfortable writing clauses in FOL syntax but reading them as sets of literals, and also identifying the empty clause with `false`.

Using Vampire to solve the `greed.p` problem we get the output shown in Figure 1. It reports a `Theorem` status because a refutation was found. The output first shows the full clausification process and then the reasoning steps. The clausification process details when a choice of Skolem function is made, etc. It includes several transformations Vampire performs which were unnecessary for this problem, like the intermediate `flattening` rewrite. The proof then consists of `resolution` reasoning steps, the first of which corresponds to the above modus ponens example.

```
$ run_vampire greed.p
% Running in auto input_syntax mode. Trying TPTP
WARNING: function_definition_elimination only useful with equality
WARNING: equality_proxy only useful with equality
% Refutation found. Thanks to Tanya!
% SZS status Theorem for greed
% SZS output start Proof for greed
1. ! [X0] : (happy(X0) <=> ? [X1] : loves(X0,X1)) [input]
2. ! [X0] : (rich(X0) => loves(X0,money)) [input]
3. rich(giles) [input]
4. happy(giles) [input]
5. ~happy(giles) [negated conjecture 4]
6. ~happy(giles) [flattening 5]
7. ! [X0] : (loves(X0,money) | ~rich(X0)) [ennf transformation 2]
8. ! [X0] : ((happy(X0) | ! [X1] : ~loves(X0,X1)) & (? [X1] : loves(X0,X1) | ~happy(X0))) [nnf transformation 1]
9. ! [X0] : ((happy(X0) | ! [X1] : ~loves(X0,X1)) & (? [X2] : loves(X0,X2) | ~happy(X0))) [rectify 8]
10. ! [X0] : (? [X2] : loves(X0,X2) => loves(X0,sK0(X0))) [choice axiom]
11. ! [X0] : ((happy(X0) | ! [X1] : ~loves(X0,X1)) & (loves(X0,sK0(X0)) | ~happy(X0))) [skolemisation 9,10]
13. ~loves(X0,X1) | happy(X0) [cnf transformation 11]
14. ~rich(X0) | loves(X0,money) [cnf transformation 7]
15. rich(giles) [cnf transformation 3]
16. ~happy(giles) [cnf transformation 6]
17. loves(giles,money) [resolution 14,15]
18. happy(giles) [resolution 17,13]
19. $false [resolution 18,16]
% SZS output end Proof for greed
% ------------------------------
% Version: Vampire 4.7 (commit )
% Linked with Z3 4.8.7.0
% Termination reason: Refutation

% Memory used [KB]: 511
% Time elapsed: 0.002 s
% ------------------------------
% ------------------------------
```

Figure 1: Solving the `greed.p` problem

The proof ends when Vampire finds the empty clause. If we trace the steps backwards, discarding no-op steps, we obtain the tree of the proof by refutation. This is shown in Figure 2. The leaves of the tree are the three premises (1, 2, and 15), the negated conjecture (16), and the choice of Skolem function (10).

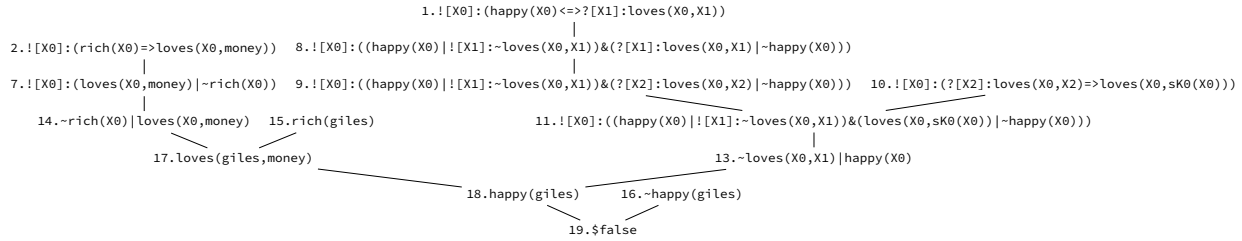


Figure 2: A solution to the `greed.p` problem

```
{
    "vampire_run": {
        "-ep": "RSTC",
        "-updr": "off",
        "-fde": "none",
        "-nm": 0,
        "-s": 1,
        "-awr": "1:1",
        "-sa": "discount",
        "-fsr": "off",
        "-av": "off"
    },
    "formulas": {
        "![X0]:(happy(X0)<=>?[X1]:loves(X0,X1))": 1,
        "![X0]:(rich(X0)=>loves(X0,money))": 2,
        "![X0]:(loves(X0,money)|~rich(X0))": 7,
        "![X0]:((happy(X0)|![X1]:~loves(X0,X1))&(?[X1]:loves(X0,X1)|~happy(X0)))": 8,
        "![X0]:((happy(X0)|![X1]:~loves(X0,X1))&(?[X2]:loves(X0,X2)|~happy(X0)))": 9,
        "![X0]:(?[X2]:loves(X0,X2)=>loves(X0,sK0(X0)))": 10,
        "![X0]:((happy(X0)|![X1]:~loves(X0,X1))&(loves(X0,sK0(X0))|~happy(X0)))": 11,
        "loves(X0,sK0(X0))|~happy(X0)": 12,
        "~loves(X0,X1)|happy(X0)": 13,
        "~rich(X0)|loves(X0,money)": 14,
        "rich(giles)": 15,
        "~happy(giles)": 16,
        "loves(giles,money)": 17,
        "happy(giles)": 18,
        "$false": 19
    },
    "proof": [
        19,
        [18,
            [17, [14, [7, [2]]], [15]],
            [13, [11, [9, [8, [1]]], [10]]] ],
        [16]
    ],
    "resolution": {
        "active": [16, 15, 13, 14],
        "passive": [12],
        "unprocessed": [17],
        "premises": [14, 15],
        "conclusion": 17,
        "mgu": [
            {"X0": "giles"},
            {}
        ]
    }
}
```

Figure 3: A proof specification for the `greed.p` problem

Your GitLab repo has a Python script called `spec2latex.py` that will help you make proof trees while solving the lab exercises. It reads a JSON specification of the proof that defines:

- An object `vampire_run` with the command-line arguments necessary to obtain the proof with Vampire.
- An object `formulas` with a dictionary mapping TPTP formulas to proof tree node numbers.
- An array `proof` that recursively specifies the tree (as a Lisp-style list of lists) of number nodes making the refutation.
- An object `resolution` with full details of a resolution step. This will be explained in the next section.

By default the script outputs the LaTeX code that creates the proof tree, but it can optionally generate the PDF directly if you have a working installation of TeX Live.

The tree in Figure 2 was made using the JSON specification in Figure 3 by running the commands

```
$ ./spec2latex.py greed.json
$ pdflatex greed.tex
```

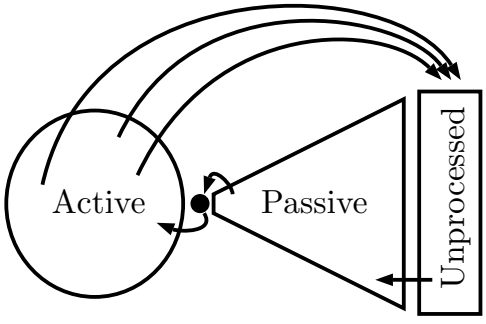
Note that in this proof specification `vampire_run` simply has the default arguments that the `run_vampire` wrapper sets.

To answer some of the lab exercises you will need to make JSON specifications (like that of Figure 3) for proofs that you construct using Vampire. Your GitLab repo includes a file called `template.json` which you can adapt to make specifications for your proofs.

Understanding Vampire's algorithm

You should aim to complete the activities in this part by the end of your assigned lab session. The goal is to practise the fourth learning objective on a toy example.

Vampire runs a given-clause saturation algorithm that we'll discuss in the lectures. It works by iteratively selecting a clause from a set of *passive* clauses, adding it to a set of *active* clauses, and performing all inferences between the selected clause and the active set, which are temporarily cached as an *unprocessed* set of clauses. The following diagram illustrates the flow of clauses from the currently unselected passive set to the currently selected active set, and the caching of unprocessed inferences.



We can ask Vampire to show the saturation process to see the order in which clauses are selected and which clauses are produced by activating (selecting) them. The option `--show_active on` prints out the clauses Vampire selects for activation. The option `--show_new on` prints out all of the clauses produced by inferences. The option `--proof_extra free` includes additional information about clauses in the output. Solving the `greed.p` problem with these options we get the output shown in Figure 4 before the refutation proof of Figure 1.

```
$ run_vampire greed.p --show_active on --show_new on --proof_extra free
% Running in auto input_syntax mode. Trying TPTP
WARNING: function_definition_elimination only useful with equality
WARNING: equality_proxy only useful with equality
[SA] new: 13. happy(X0) | ~loves(X0,X1) [cnf transformation 11] {a:0,w:5,thAx:0,allAx:2,thDist:-2}
[SA] new: 12. loves(X0,sK0(X0)) | ~happy(X0) [cnf transformation 11] {a:0,w:6,thAx:0,allAx:2,thDist:-2}
[SA] new: 14. loves(X0,money) | ~rich(X0) [cnf transformation 7] {a:0,w:5,thAx:0,allAx:1,thDist:-1}
[SA] new: 15. rich(giles) [cnf transformation 3] {a:0,w:2,thAx:0,allAx:1,thDist:-1}
[SA] new: 16. ~happy(giles) [cnf transformation 6] {a:0,w:2,goal:1,thAx:0,allAx:1,thDist:-1}
[SA] active: 16. ~happy(giles) [cnf transformation 6] {a:0,w:2,nSel:1,goal:1,thAx:0,allAx:1,thDist:-1}
[SA] active: 15. rich(giles) [cnf transformation 3] {a:0,w:2,nSel:1,thAx:0,allAx:1,thDist:-1}
[SA] active: 13. ~loves(X0,X1) | happy(X0) [cnf transformation 11] {a:0,w:5,nSel:1,thAx:0,allAx:2,thDist:-2}
[SA] active: 14. ~rich(X0) | loves(X0,money) [cnf transformation 7] {a:0,w:5,nSel:1,thAx:0,allAx:1,thDist:-1}
[SA] new: 17. loves(giles,money) [resolution 14,15] {a:1,w:3,thAx:0,allAx:2,thDist:-2}
[SA] active: 17. loves(giles,money) [resolution 14,15] {a:1,w:3,nSel:1,thAx:0,allAx:2,thDist:-2}
[SA] new: 18. happy(giles) [resolution 17,13] {a:2,w:2,thAx:0,allAx:4,thDist:-4}
[SA] active: 12. ~happy(X0) | loves(X0,sK0(X0)) [cnf transformation 11] {a:0,w:6,nSel:1,thAx:0,allAx:2,thDist:-2}
[SA] active: 18. happy(giles) [resolution 17,13] {a:2,w:2,nSel:1,thAx:0,allAx:4,thDist:-4}
[SA] new: 19. $false [resolution 18,16] {a:3,w:0,goal:1,thAx:0,allAx:5,thDist:-5}
[SA] new: 20. loves(giles,sK0(giles)) [resolution 18,12] {a:3,w:4,thAx:0,allAx:6,thDist:-6}
% Refutation found. Thanks to Tanya!
```

Figure 4: Saturation for the `greed.p` problem

The saturation process that Vampire followed while solving `greed.p` is shown in lines 5 to 20 of Figure 4. The first 6 lines report the order in which the **`cnf`** clauses for the problem passed from the unprocessed to the passive set. After this Vampire starts activating clauses, but no inferences are possible until formula 14 becomes active and it derives the clause `loves(giles,money)` in the way described above by the modus ponens example of the previous section. This resolution step involves the premises (14 and 15), the most general unifier $\left[x \mapsto \text{giles}\right]$, and the conclusion (17). The state of the given-clause saturation process at this point is given by the list of passive (only 12 is unselected), active (16, 15, 13, and 14 were selected), and unprocessed clauses (17 was the only clause generated by the selection of 14).

All this information is collected in the `resolution` object of the JSON specification in Figure 3. Note that the representation of the mgu must a pair of JSON objects: the first fixes a map for variables in the LHS premise, and the second a map for variables in the RHS premise. In this example, the LHS formula 14 has one free variable `X0` and the RHS formula 15 has no free variables at all. You may find situations in which resolution premises have free variables in common that their most general unifier will map differently.

Now continue following the given-clause algorithm for yourself. Con you give the full details of the resolution step which obtains formula 18? How about the one that derive the empty clause 19? You will need to construct these when answering some of the lab exercises.

Important: Vampire has various options to show additional information about its state. However, remember that Vampire's algorithm includes many refinements that go beyond the basic given-clause algorithm explained above. In the lab exercises you will be asked to track the state of this basic given-clause saturation process (for the clause selection approach that you configure using Vampire's options). In general, this will be different from the one Vampire reports when run with the `--show_everything on` option for example.

To help you understand Vampire's output better, we'll briefly explain the full meaning of line 15 in Figure 4. 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 `17` is the id number of the clause — clauses are numbered as they are produced. Then we have the clause `loves(giles,money)` in clausal form. The next bit `[resolution 14,15]` describes how the clause was generated, here it was by resolution from clauses `14` and `15`. Finally, there is the extra information string `{a:1,w:3,nSel:1,thAx:0,allAx:2,thDist:-2}` giving values that are used for clause and literal selection, etc.

The important fields for understanding Vampire's saturation strategy are:

- the age of the clause `a`,
- the weight of the clause `w`,
- the number of selected literals in the clause `nSel` (which are always the leftmost literals in active clauses), and
- the `goal` if the clause is derived from the negated conjecture.

Using the command line option `-s` we can control how Vampire does literal selection. The default value of `run_vampire` corresponds to *ordered* resolution, which means only maximal literals are chosen as atoms for resolution. If we use `-s 0` we get *unordered* resolution, which allows all literals to be chosen during resolution. You can control how Vampire performs clause selection using the command line option `-awr` which controls the ratio between selecting clauses by age and by weight. For example, if you specify `-awr 10:1` then for every 10 clauses selected by age Vampire will select one by weight.

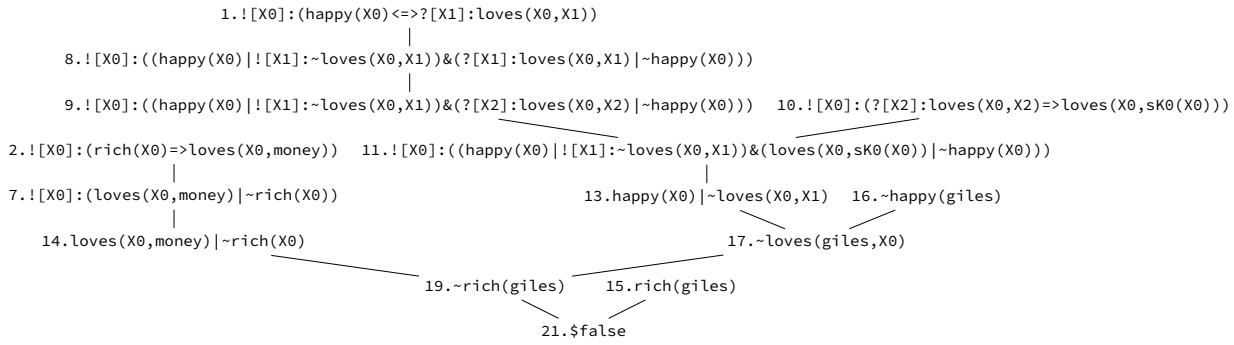


Figure 5: Solving `greed.p` with `-s 0 -awr 1:10`

Figures 5 and 6 show how changing clause and literal selection strategies results in different proof trees for the `greed.p` problem. Indeed, each generates the empty clause by resolving on a different literal: `happy(giles)` in Figure 2, `rich(giles)` in Figure 5, and `loves(giles,money)` in Figure 6. This last clause, which we have been tracking since the previous section, isn't even present in the second tree.

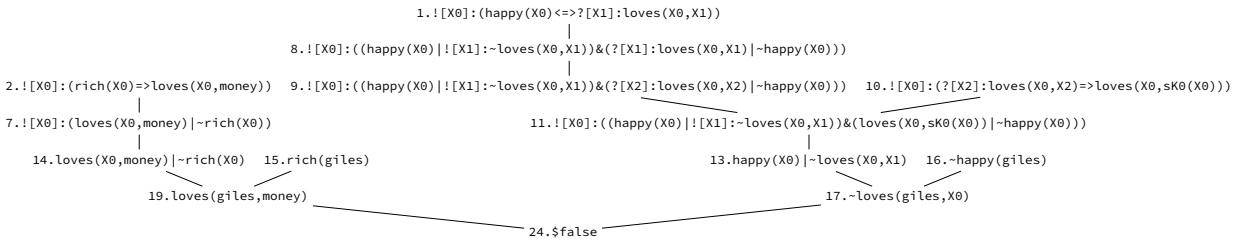


Figure 6: Solving `greed.p` with `-s 0 -awr 10:1`

In the lectures we will study different clause and literal selection strategies. In this lab you'll be asked to **experiment** with these and explore the results. You need to record the various `-s` and `-awr` values that you use in the `vampire_run` object of your proof specifications.

In preparation for the lab exercises make sure that you are able to reproduce the proof trees in Figures 5 and 6 before the end of your lab session. In your JSON specifications, add the details for a resolution that only appears in that particular tree, including a complete description of the saturation process state, as appears in Figure 3. This will allow you to better understand why we arrive at different proofs.