Python Support for Syllogistic Logic: Inference and Counter-Models

Project for Annie Pompa

item	target date, tentative
The basic syllogistic inference engine of Section 1 including proofs in list form (probably using my Sage code for that)	December 1
Addition of partial functions	after finals
Symbols of higher arity; this might mean re-working the the basic syllogistic engine	not so important: could be done any time next semester
Model construction for basic syllogistic logic I will help on this, and the main hard point will be to have a nice output	February 1
Model construction for cardinality logic (this is what is implemented on CoCalc) done with my help, probably	March 1
Model construction for logics with verbs with my help, probably (this also has been done in Haskell)	April
I'll try to find a research project that you could do based on everything above. One possibility: integration with SAT solvers. Another: new logics that haven't been studied.	end of winter 2021 semester

1 The Basic Syllogistic Inference Engine

This section describes the basic *syllogistic inference engine*. It is the main part of both *proof search* and *counter-model generation* in a family of syllogistic logics.

1.1 Definitions concerning rules

A *variable* is one of the symbols u, v, w, x, y, z. (Actually, the variables could be anything, and we might need more than 6 of them. My guess is that 20 is the more than anyone would want in practice.)

A *tag* is one of the symbols a, e, i, o. As with variables, the tags could be anything. It would be confusing to use the same letter for variables and for tags, but technically this could work out. *It would probably better to use numbers as tags.*

A *statement* is a tuple (r, x, y), where r is a tag and x and y are variables.

A *premise list* is a list $[s_1, \ldots, s_k]$ of statements. The empty list is a premise list. Usually our premise lists have length 2, and occasionally they are longer. I'd say that 6 should be more than we ever want.

An *inference rule* is a triple consisting of a rule name, a premise list and a statement.

1.2 Examples

Example 1. Here are examples of rules:

1. The syllogistic rule (barbara):

$$(\text{barbara}, [(a, x, y), (a, y, z)], (a, x, z))$$

   The premise list is $[(a, x, y), (a, y, z)]$, and the *conclusion* is (a, x, z).

2. The syllogistic rule (darii):

$$(\text{darii}, [(a, x, y), (i, x, z)], (i, y, z))$$

3. The rule (axiom) has an empty premise list:

$$(\text{axiom}, [\,], (a, x, x))$$

It is the only rule with an empty premise list, and it is thus kind of an outlier. All of the other rules have the nice feature that every variable in the conclusion occurs in one of the premises.

4. Here's a rule I just made up

$$(\text{junk}, [(e, x, y), (e, y, x), (o, y, x)], (u, x, y))$$

I used e, o, and u as tags.

1.3 Definitions concerning databases

A *database* in this discussion is as a set (or list) of numbers $[0, 1, \ldots, K]$ called the *universe*, and a set of *tag facts* (t, m, n), where t is a tag and m and n are numbers in the list.

Example 2. Here is an example. Let's take the universe to be $[0, 1, 2, 3, 4]$ and the tag facts to be

$$[(a, 0, 2), (a, 2, 3), (a, 2, 4), (a, 4, 1), (e, 3, 3), (i, 0, 3), (i, 3, 2)]$$

1.4 Inputs

There are two kinds of inputs:

1. A list of rules.
2. A database in the sense above.
3. Another tag fact called the *target*.

Important note The inference engine described here is going to be the central module inside a bigger system. There will eventually be a front end and a back end. In fact, this engine will get used in several different ways in the bigger system. There is no real reason to arrange for convenient inputs at this stage. And there's no reason to have very readable outputs, either.

Example 3. We could take the list [barbara, darii, axiom, junk] from Example 1, and the database from Example 2. And for the target, we could take $(a, 2, 1)$.

1.5 Outputs

We want to *whether or not the target could be generated using all applications of rules to the database*. If the target can be generated, then we want either a proof tree or a proof list. If it can't be generated, we want the output to say this.

I won't define "generated" formally, since it's part of the problem. But I will give examples, following on Example 3. I can of course give you suggestions on how you would implement this, using either matrices (as in the CoCalc implementation) or (better) hash tables, as in your idea.

Example 4. If the user inputs the list [barbara, darii, axiom, junk] from Example 1, the database from Example 2, and the target $(i, 2, 3)$, we should output the tree below

$$\dfrac{\dfrac{(a,0,2) \quad (a,2,3)}{(a,0,3)}\ \text{barbara} \qquad (i,0,3)}{(i,2,3)}\ \text{darii}$$

A little more explanation. In the use of (barbara), we took $x = 0$, $y = 2$, and $z = 3$. We will want to apply all of the rules in all possible ways to the database, and we add the new facts to the database as we go. And we want to do this "forever"; that is, we do it repeatedly until we stop generating new facts to put in the database.

It would be nice if the tree were shown on the screen, but I have never gotten this to work. And with a wide tree, this is impossible anyways. In that case, we would want the output to be a proof shown as a list:[1]

1	$(a, 0, 2)$	premise
2	$(a, 2, 3)$	premise
3	$(i, 0, 3)$	premise
4	$(a, 0, 3)$	barbara 1, 2
5	$(i, 2, 3)$	darii 4,3

Example 5. If the user inputs the list [barbara, darii, axiom, junk] from Example 1, the database from Example 2, and the target $(a, 4, 4)$, we should

[1] Part of my existing Python code could be used to convert the tree representation of a proof to the list representation; that is, I have worked out how this goes. I found it tricky to work out the algorithm for attaching line numbers to points in the proof tree.

output the tree below

$$\frac{}{(a,4,4)}\ \text{axiom}$$

Note that there is nothing above the line. As a list, this would be

$$1 \quad (a,4,4) \quad \text{axiom}$$

There is just one rule used, the one called "axiom", and we used it with $x = 4$.

Example 6. If the user inputs the list [barbara, darii, axiom, junk] from Example 1, the database from Example 2, and the target $(e, 3, 0)$, we should output that $(e, 3, 0)$ cannot be generated from the database using the rules. That is, we should generate everything that we possibly could get from the database, and then once we are done we can look inside and see that $(e, 3, 0)$ is not there.

Important note (continued) What I am describing here is a bit of a back end for the inference engine. But it is not the "ultimate" back end because that would make use of sentences in English. So in a sense what I am asking for here is a *preliminary back end* that will be improved upon later.

1.6 Intermediate step: a front end and a back end for proof search in classical syllogistic logic

Section 2 is the next real step in the overall project. But as I read things over, I see that you won't understand what is going on just on the basis of the programs that are needed. That's because what you did already is really the *middle* of an end-to-end system on *proof search in (my version of) classical syllogistic logic* The year-long project overall is about *extend syllogistic reasoning*, and we'll get to that in due course.

For classical syllogistic reasoning, here's what we want:

1. The user should call a program that could be called S (or something like it); S is for syllogistic. This program would have a dedicated set of

rules:

$$\begin{array}{l}
(\text{axiom}, [\,], (a, x, x)) \\
(\text{barbara}, [(a, x, y), (a, y, z)], (a, x, z)) \\
(\text{darii}, [(a, x, y), (i, x, z)], (i, y, z)) \\
(\text{some1}, [(i, x, y)], (i, x, x)) \\
(\text{some2}, [(i, x, y)], (i, y, x)) \\
(\text{zero}, [(a, x, n(x))], (a, x, y)) \\
(\text{one}, [(a, n(x), x)], (a, y, x)) \\
(\text{anti}, [(a, x, y)], (a, n(y), n(x))) \\
(\text{x}, [(a, x, y), (i, n(x), y], \text{any})
\end{array}$$

The new feature here is the function symbol $n()$ and I'll explain this in the course of these pages. The letter n stands for "negation", and we think of it like the English word *non*. In addition the rule (x) at the bottom requires special handling that I'll also discuss.

2. When the user calls S, they don't have to input these rules. The rules will be part of the program. But the user gets to input a *premise list* consisting of English sentences like

   'all a are c', 'some non-e are c', 'all c are non-d' (1)

   We could even get fancy and allow the user to use names of (say) animals, as in

   'all cats are dogs', 'some non-horses are cats', 'all dogs are reptiles'

   For now, I'll just use letters as in (1).

   The user also inputs a *target* in the same form. This time, it could be 'some d are e'.

   By the way, these rules are part of the CoCalc implementation, as is this front end. So if you play with proof search on CoCalc, you'll see what is to be done here. The feature of CoCalc that we are not touching yet is the *model-builder*, and this will be for next semester.

3. The first thing that S does is to extract the raw variables from the input, and to make a dictionary from it. For the input in (1) the variables are a, b, c, d, and e (without the word *non*), and the dictionary would be

```
{ 0:a, 1:b, 2:c, 3:d, 4:e }
```

You might want to have quotes on the variables, this is not important.

4. Next, S adds *formal negations* by expanding the dictionary to

```
{0:a, 1:b, 2:c, 3:d, 4:e, 5:non-a, 6:non-b,7:non-c, 8:non-d, 9:non-e}
```

5. Make a database with universe $[0, 1, \ldots, 9]$ and with the extra information that n works as follows:

i	$n(i)$
0	5
1	6
2	7
3	8
4	9

i	$n(i)$
5	0
6	1
7	2
8	3
9	4

The tag facts in this universe are the translations of the premise list from (1), using a for *all*, and i for *some*, and being careful about n:

$$[(a, 0, 2), (i, 9, 2), (a, 2, 8)]$$

We also translate the target 'some d are e' to $(i, 3, 4)$.

6. Now we run the program from before, but we have to again be clear about n. We should get that there is a no proof.

   But if we ask for a different target, say the translation of 'all d are non-a', namely $(a, 3, 5)$, we should get a proof, namely the one below:

1	$(a, 0, 2)$	premise
2	$(a, 2, 8)$	premise
3	$(a, 0, 8)$	barbara 1,2
4	$(a, 3, 5)$	anti 3

   The main thing to look at is the treatment of n in the application of anti at the bottom.

7. We would finally like to print things out back in English, not as a list of tag facts. We want:

1	all a are c	premise
2	all c are non-d	premise
3	all a are non-d	barbara 1,2
4	all d are non-a	anti 3

8. The rules for this particular logic have (x), where x stands for *Ex falso quodlibet* or *Ex contradictione quodlibet* . The rule is related to *contradictions* in logic. It says that *from a contradiction, the reasoner can have anything she wants.* The way I stated (x) above basically says this: it says that *any* tag fact should follow from a contradiction.

   There are several ways that this rule can fit in a syllogistic proof engine like the kind we are building. These ways are related, and I'm not yet sure which one we'll want in the ultimate system. For now, we want support for contradictions in the following way. A database of tag facts is *inconsistent* if (x) can be applied. That is, if there are tag facts corresponding to the English *all x are y* and *some x are non-y*. These facts don't have to be literally in the input database, they can be derivable. When a user enters a premise list which is inconsistent, we want the program for S to say "this list of premises is inconsistent, and here's why". And then the program should produce a proof tree that ends with the contradiction in the last two lines, showing that with one more application of (x), we can derive anything. This is how the implementation on CoCalc works, and you might look back at the CoCalc to try it out.

2 Next Task: partial function symbols in rules and in databases

Once we have a system which can inputs and make outputs as described above, there are several next steps. First, we want *function symbols* on variables interpreted as *partial functions* in our databases. For example, we might like to have a function symbol n. That we could use in our rules. So one of the rules might be

$$(\text{more-rule}, [(a, x, y), (o, y, x)], (i, x, n(y)))$$

All of the function symbols that we need are *unary* (one-place) symbols. But we might need to nest these symbols, as in

$$(\text{junk-rule}, [(a, x, n_2(n_1(y))), (o, y, x)], (i, x, n_1(n_2((y))))$$

The presence of "undefined" is what makes for *partial functions*. I want to "reify the undefined" here by adding a new object $\perp$ to the universe. For example, in a database, we might want some (partial) interpretation of n such as

item q	$n(q)$
0	1
1	0
2	$\perp$
3	2
4	$\perp$
5	4

Note that we don't list $\perp$ on the left in the chart. That's because $\perp$ is treated in a special way. It's really thought of as "exception", or "strange". So when we instantiate rules, we don't want to take the value $\perp$ on any of the variables. And $\perp$ doesn't appear in any tag facts, ever.

And then we'll want to instantiate this kind of rule the right way. **more to come here**.

Example 7. Here is a list of three rules:

$$[\text{barbara}, \quad (\text{r1}, [(a, f(y), f(x))], (a, x, y)), \quad (\text{r2}, [(a, g(y), g(x))], (a, x, y))]$$

As before barbara is the rule

$$(\text{barbara}, [(a, x, y), (a, y, z)], (a, x, z))$$

So f and g are partial function symbols.

For our database, we take the universe to be $[0, 1, 2, 3, 4, 5]$ and a list of the following two tag facts:

$$[(a, 0, 1), (a, 5, 1)]$$

We interpret the partial function our universe as in the table below

item q	$f(q)$	$g(q)$
0	2	$\perp$
1	3	$\perp$
2	$\perp$	4
3	$\perp$	5
4	$\perp$	$\perp$
5	$\perp$	$\perp$

Again, our database consists of the universe, the list of tag facts, and the interpretations of the two partial function symbols.

Let's take the target query to be $(a, 4, 1)$. Then our system should produce a proof tree as shown below:

$$\dfrac{\dfrac{\dfrac{(a,0,1)}{(a,3,2)}\, r1}{(a,4,5)}\, r2 \qquad (a,5,1)}{(a,4,1)}\ \text{barbara}$$

This is a tree with two leaves, namely the tag fact $(a, 0, 1)$ and the tag fact $(a, 5, 1)$. In the application of $r1$, we took $x = 0$ and $y = 1$, so that $f(x) = 2$ and $f(x) = 3$.

In the application of $r2$ to $(a, 3, 2)$, we took $x = 3$ and $y = 1$, so that $g(x) = 5$ and $g(x) = 4$.

In the application of (barbara) at the bottom, we took $x = 4$, $y = 5$, $z = 1$.

Example 8. We could take the same rules as in the previous example, and also the same database. Then we could change the target to $(a, 4, 3)$. In this case, the system should reply that $(a, 4, 3)$ is not derivable. And it should do so after generating all of the possible proof trees determined from the database and the rules.

In fact, we want to be able to take our rules and database and ask for *all derivable tag facts*. In this example, we should get the list below:

$$[(a, 0, 1), (a, 5, 1), (a, 3, 2), (a, 4, 5), (a, 4, 1)]$$

3 Step 3: tags with "arity 3" or "arity 4"

We also want tags with arity > 1, as in

$$([(u, x, y, z), (u, y, z, z), (a, x, z)], (e, x, y))$$

In what went before, we only had "arity 2" tags.

4 Model building: the All Logic

4.1 Front End: mostly done already