Exercise 9:

Deductive Proofs of First-Order Logic Formulae

In this exercise, we will develop the notion of formal proofs for first-order logic. As in the case of propositional logic, we will have axioms and inference rules, but we will now need to handle all of the new elements of first-order logic. We will avoid duplicating the effort that we made in dealing with propositional logic by defining as axioms all tautologies, which as we have seen can be proven from a few simple axioms using propositional logic.

Schemata

Recall that in propositional logic, we could have a formula like '(x|~x)' as an axiom, with the understanding that we can plug in any Boolean expression for x. Thus, for example, substituting '(p→q)' for x gives that the formula '((p→q)|~(p→q))' is an instance of this axiom. We will need a similar mechanism for predicate logic, but here we will make the rules of what exactly can be substituted and where explicit. While in propositional logic we either had assumptions that have to be taken completely literally without any substitutions on one extreme, or axioms where essentially any symbol could be consistently replaced with any formula when using the axiom on the other extreme, in first-order logic we will have the full range in between these two extremes. For example we may wish to write an assumption such as 'plus(c,d)=plus(d,c)', where c and d could be replaced by any terms but plus cannot be replaced by any other function. As another example, we will also wish to represent *axiom schema* that stand for a collection of axioms of a certain type. In traditional mathematical textual proofs, for example, we can say things like "for any formula ϕ, any term τ, and any variable x, the following is an axiom: '$(\forall x[\phi(x)]\rightarrow\phi(\tau))$', and we will wish to capture such statements as axioms as well. In our computerized representation, we will use simple relation names as **templates** for formulae, simple constant names as **templates** for terms, and simple variable names as **templates** for variable names, so we will represent this axiom schema as '(∀x[R(x)]→R(c))' while explicitly stating which symbols (if any) represent templates ('R' and 'c', and 'x', in this case) and which (if any) should be taken literally.

Before we get into the exact details of schemata, we start by implementing the notion of *substitution* in a `Term` or `Formula`, where a constant name or variable name is "replaced" by some other term.

Task 1. Implement the missing code for the method `substitute(self, substitution_map)` of class `Term`, which takes a dictionary that maps variable names and constant names to `Term` objects, and returns a `Term` object obtained by having, for each constant name or variable name `k` that is a key of `substitution_map`,

the term `substitution_map[k]` substituted for every occurrence of `k` in the current term (all substitutions are performed simultaneously). (Constant names and variables names that are not keys of `substitution_map` remain as is.) The current term is not modified of course (recall from Exercise 7 that we treat all `Term` objects as immutable). For example, for `term = Term('f', [Term('x'), Term('c'), Term('plus', [Term('x'), Term('z')])])`, calling

```
term.substitute({'x':Term('0'), 'c':Term('times',Term('w'),Term('1'))})
```

should return a `Term` object representing 'f(0,times(w,1),plus(0,z))'.

Task 2. Implement the missing code for the method `substitute(self, substitution_map)` of class `Formula`, which takes a dictionary that maps constant names and variable names to `Term` objects, and returns a `Formula` object obtained from the current formula by simultaneously performing the following substitutions: for each constant name `c` that is a key of `substitution_map`, the term `substitution_map[c]` is substituted for every occurrence of `c`, and for each variable name `v` that is a key of `substitution_map`, the term `substitution_map[v]` is substituted for every *free* occurrence of `v`. (Constant names and variables names that are not keys of `substitution_map` remain as is, and so do bound occurrences of variable names that are keys of `substitution_map`.) The current formula is not modified of course (recall from Exercise 7 that we treat all `Formula` objects as immutable).

We now move on to defining (and programming) exactly what a *formula schema* is and which formulae are instances of it. The file `predicates/proofs.py` defines a Python class `Schema` that represents a formulae schema as a regular first-order formula together with the set of the elements in the formula that are to be viewed as templates. Three types of syntactic elements can potentially serve as templates: constant names, variables names, and relation instantiations.

A constant name that is specified as a template can serve as a placeholder for any term. Thus, for example, the schema `Schema('c=c', {'c'})` has as specific instances all of the following formulae: '0=0', 'x=x', 'plus(x,y)=plus(x,y)'.

A variable name that is specified as a template can serve as a placeholder for any variable name. Thus, for example, the schema

```
Schema('(times(x,0)=0&Ax[Ey[plus(x,y)=0]])', {'x'})
```

has the following as an instance: '(times(z,0)=0&∀z[∃y[plus(z,y)=0]])'. As this example shows, all occurrences of the template variable name should be replaced, including free and bound occurrences, as well as occurrences that immediate follow a quantification.

A relation that is specified as a template can serve as a placeholder for an arbitrary formula, possibly with "parameters." For example, the schema

```
Schema('(Ax[(R(x)->Q(x))]->(Az[R(z)]->Aw[Q(w)]))', {'R', 'Q'})
```

can be instantiated with 'R(v)' (where 'v' is a formal parameter name, i.e., a placeholder for the parameter of R when instantiating the above schema) defined as 'v=7' and with 'Q(v)' defined as 'T(y,v)' to obtain the following instance of this schema: '(∀x[(x=7→T(y,x))]→(∀z[z=7]→∀w[T(y,w)]))'. Note that like any other relation, a relation template that appears in a schema may have any number of arguments, including zero.

Note that in our instantiation of 'Q(v)' as 'T(y,v)' above, we allow the instantiated formula to use variable symbols ('y', in this case) beyond its formal parameters, and these just get copied into the schema instance as is. In order to avoid unintended (and logically incorrect) quantifications, this, however, is only allowed for variables that *do not get bound by a quantifier in the resulting schema instance*. Thus, for example, if we look at the schema `Schema('(R()->Ax[R()])', {'R'})`, then we are allowed to substitute for R() the formula 'Q(y)' to get '(Q(y)→∀x[Q(y)])', but we may not substitute 'Q(x)' for it, so '(Q(x)→∀x[Q(x)])' is *not* an instance of this schema (nor is it a logically valid statement), since the free variable x in Q(x) would get bound by the universal quantifier if such a substitution were to be performed.

Let us look more closely at the process of, e.g., taking the schema given in the first example above, `Schema('(Ax[R(x)]->R(c))', {'c', 'R'})`, and instantiating it with 's(1)' for 'c' and with '(∃z[s(z)=v]→Q(v))' for 'R(v)'. For the first occurrence of 'R' in the schema we substitute 'x' for 'v' in the replacement for 'R(v)' and get '(∃z[s(z)=x]→Q(x))', while for the second occurrence we substitute 's(1)' for 'v' (since we substitute 's(1)' for 'c', which gets substituted for 'v' in the replacement for 'R(v)') and get '(∃z[s(z)=s(1)]→Q(s(1)))', together getting the instance '(∀x[(∃z[s(z)=x]→Q(x))]→(∃z[s(z)=s(1)]→Q(s(1))))' of this axiom schema (which is indeed a logically valid statement).

In the process of substitution, we replace every *free* occurrence of the formal parameter 'v' (of 'R(v)') with the term given as a parameter to the relation template 'R' in the schema formula. We emphasize that bound occurrences of the formal parameter 'v' are not replaced: in case there is an internal quantification over 'v' in the formula for 'R', then, since we assume "local scope,"[1] an occurrence of 'v' inside the internal quantification refers to the inner quantified 'v' rather to the "outer" version to be substituted. For example, instantiating the same axiom schema with '0' for 'c' and with '∃z[(s(z)=v&∀v[(~v=z→~s(v)=s(z))])]' for 'R(v)', we would get

'(∀x[∃z[(s(z)=x&∀v[[(~v=z→~s(v)=s(z))]])]]→∃z[(s(z)=0&∀v[(~v=z→~s(v)=s(z))])])'

The constructor of the class `Schema` accepts a first-order formula and a set of elements in the formula that are to be treated as templates. The default value for the set of templates is the empty set, meaning a schema with only one possible instance: the given formula. The most important functionality of this class is to *instantiate* the current schema according to a given *instantiation map*. The two `substitute()` methods that you have implemented in Tasks 1 and 2 above handle most of the burden of instantiating constant and variable templates. We still need to focus on handling the instantiation of relation templates, and correctly combining it with the instantiation of constant and variable templates. The helper method that you will implement in the following task does most of this work.

Task 3. Implement the missing code for the static method `instantiate_formula` of class `Schema`. This recursive method takes four arguments:

1. A `formula`.
2. A dictionary `constants_and_variables_instantiation_map` that maps each constant template name and variable template name to a `Term` that should replace the corresponding template in the original Formula. Constant names may

[1] Recall from Exercise 7 that we interpret a variable name in any specific context by the inner-most scope surrounding it that defines this variable.

be mapped to any Term, while variable names may only be mapped to a Term whose root is a variable name.

3. A dictionary `relations_instantiation_map` that maps to each relation template name to a pair that specifies how this relation template should be instantiated. The first entry of this pair is a list of variable names that serve as formal parameters (as defined above), and the second is a `Formula` object over these variables names (as well as possibly other variables names, constants names, other relation names, etc.) to be substituted for this relation template. You may assume that the number of parameters in every "call" to any relation template in `formula` is the same as defined in the `relations_instantiation_map` key for that relation.

4. A set `bound_variables` of variable names that are to be treated as being quantified by outer layers of the recursion. The implication is (see below) that a template relation may not be instantiated to an expression that has free variables (other than the formal parameter names) in this set, just like (see below) it may not be instantiated to an expression that has free variables (other than the formal parameter names) that get quantified when plugged into `formula`.

The method returns the `Formula` obtained from the given `formula` by simultaneously performing all of the substitutions specified by the given variables-and-constants instantiation map and by the given relations instantiation map. The original formula is not modified, of course. **Example:** calling this method with `formula=Formula.parse('(Ax[R(x)]->R(c))')`, with `constants_and_variables_instantiation_map={'c':Term.parse('f(0)'), 'x':Term('z')}`, with `relations_instantiation_map={'R':(['v'], Formula.parse('v=y'))}`, and with `bound_variables=set()`, should return a `Formula` object whose string representation is `'(Az[z=y]->f(0)=y)'`.

The method should raise a `Schema.BoundVariableError` exception (defined for you in `predicates/proofs.py`) if any free variable in any formula that is given as a second entry of a value in `relations_instantiation_map` and that the method substitutes for a relation template, gets bound when substituted into `formula` or is in the set `bound_variables`. **Examples:** for `formula=Formula.parse('Ax[(Q(z)->R(x))]')`, for `constants_and_variables_instantiation_map={}`, and for `relations_instantiation_map={'Q': (['v'], Formula.parse('x=v'))}`, an exception will be raised, since if `'Q(z)'` is substituted by `'x=z'` then x becomes bound. For the same arguments but with the above `relations_instantiation_map` replaced with `{'Q': (['v'], Formula.parse('y=v'))}`, an exception will be raised if and only if `'y'` is an element of the set `bound_variables` passed as the method's fourth argument. **Guidelines:** the method should naturally be implemented recursively. The two simple base cases are when `formula` is a relation instantiation of the form '$R(t_1,\ldots,t_k)$' that is not a template, or an equality of the form '$t_1{=}t_2$' (where in either, each t_i is a term). The required substitution here is simply given by the `substitute()` method of class `Formula`. The interesting base case is when `formula` is a relation instantiation of the form '$R(t_1,\ldots,t_k)$' that *is* a template that should be replaced with a formula $\phi(v_1,\ldots,v_k)$, where $(v_1,\ldots,v_k)$ are the corresponding formal parameters. In this case, one should first use the `substitute()` method of class `Term` to make the substitutions in each of the "actual arguments" $t_1,\ldots,t_k$, getting the post-substitution terms $t'_1,\ldots,t'_k$. Afterward, one should "plug" $t'_1,\ldots,t'_k$ into ϕ by using the `substitute()` method of class `Formula`, with the substitution map $\{v_1{:}t'_1,\ v_2{:}t'_2,\ \ldots,\ v_k{:}t'_k\}$. (Of course, one

should first check that the free variables of ϕ, beyond the formal parameters $v_1, \ldots, v_k$, are disjoint from the set `bound_variables` — otherwise an exception should be raised.) The recursion over the formula structure is simple, where the only nontrivial step is the case when `formula` is a quantification of the form '$\forall x[\phi]$' or '$\exists x[\phi]$'. In this case, if the quantification variable is a template then it should be replaced as specified, and (regardless of whether or not the quantification variable is a template) deeper recursion levels should take the quantification into account, i.e., the quantification variable (after any replacement) should be included in the `bound_variables` set that is passed to deeper recursion levels.

We are now ready to implement the main method of the `Schema` class.

Task 4. Implement the missing code for the method `instantiate(self, instantiation_map)` of class `Schema`, which takes a dictionary that assigns to each name in the schema template what it should be instantiated to, and returns the instantiated formula, as explained above (between Task 2 and Task 3). Templates that are not assigned any value by the given instantiation map remain as is in the returned formula. If the instantiation map attempts to instantiate a constant, variable, or relation that is not a template, then `None` should be returned. `None` should also be returned in case of an illegal relation substitution, that is, a formula to be substituted for a relation, whose free variables beyond the formal parameters of the relation template are to become bound in the schema formula. For example, for `schema=Schema('(Ax[R(x)]→R(c))', {'c', 'R'})`, calling `schema.instantiate({'c':'s(0)', 'R(v)':'Ey[Plus(v,1,y)]'})` will return a `Formula` object representing '(∀x[∃y[Plus(x,1,y)]]→∃y[Plus(s(0),1,y)])'. You may assume that the number of parameters in every "call" to any relation template in the current formula is the same as defined in the `substitution_map` key for that relation. **Guidelines:** call your solution to Task 3 with the appropriate arguments that you will derive from `instantiation_map`. Do not forget to check for illegal arguments, and to handle `Schema.BoundVariableError` exceptions.

Proofs

We can now move to defining formal deductive proofs in first-order logic. A proof gives a formal derivation of a conclusion from a list of assumptions/axioms, via a set of inference rules, where the derivation itself is a list of lines, each of which is justified by an assumption/axiom, or by previous lines via an inference rule. There are many possible variants for the set of inference rules and for the set of logical axioms, and we will use a system that has the following allowed types of justifications for each line in the proof:

1. **Assumption/Axiom**: We may, of course, use any of our assumptions/axioms in the proof, and note that as our assumptions/axioms are given as schemata, any instance of an assumed/axiomatic schema is valid. For simplicity, we do not make any programmatic distinction between assumptions and axioms, however one can think of assumptions/axioms that are schemata with templates as playing a part somewhat similar to that of the axioms from our propositional-logic proofs, while assumptions/axioms that do not have any templates can be thought of as analogous to regular assumptions in our propositional-logic proofs.

2. **Tautology**: We will allow every tautology as an axiom, where a tautology is a formula that is true when viewed as a propositional formula (see below for the precise definition). For example, '(R(x)→R(x))' is a tautology. Allowing any tautology is purely for simplicity; we could have alternatively added 14 schemata that represent the 14 axioms from Exercise 6 as assumptions/axioms (e.g., the propositional axiom I1 could be represented as `Schema('(P()->(Q()->P()))', {'P', 'Q'})`), and "inlined" the proof of any needed tautology.

3. **Modus Ponens**: We will keep *Modus Ponens*, or MP, as an inference rule. That is, from ϕ and '$(\phi \rightarrow \psi)$' we may deduce ψ.

4. **Universal Generalization**: We introduce a new inference rule: universal generalization, or UG:[2] from any formula ϕ and any variable x we may deduce '$\forall x[\phi]$'. Examples: from the formula '(R(x)→Q(x))' we may deduce '∀x[(R(x)→Q(x))]' as well as '∀z[(R(x)→Q(x))]'.

The file `predicates/proofs.py` defines the Python class `Proof` that represents such a proof, and contains the following components:

- A list of assumptions, each of them a `Schema`.
- A conclusion that is a `Formula` object.
- The body of the proof that is a list of **lines**, where each line is an object of class `Proof.Line`, which contains two parts:
  - A `Formula` object that is the statement claimed by this line.
  - The **justification** of this line from previous lines or the assumptions. The justification is a tuple, and there are exactly four possible types of justifications:
    1. A 3-tuple[3] `('A',i,instantiation_map)` that justifies a line as an instance of an assumption/axiom. Here `i` is the index of the assumption/axiom, and `instantiation_map` is a dictionary according to which this assumption/axiom (which is a schema) can be instantiated to obtain the current line. (The dictionary may be an empty dictionary if the assumption/axiom contains no templates.)
    2. A 1-tuple `('T',)` that states that the current line is a tautology.
    3. A 3-tuple `('MP',i,j)`, that justifies a line that is a *Modus Ponens* conclusion. Here `i` and `j` are indices of previous lines in the proof, where line `i` holds some formula ϕ and line `j` holds the formula '$(\phi \rightarrow \psi)$', where the current line holds the formula ψ.
    4. A 2-tuple `('UG',i)` that justifies a line that is a universal generalization of a previous line whose index is `i`.

[2]For simplicity, in our first-order logic proofs we force what we only used as a convention in our propositional-logic proofs: that whenever we can "encode" a needed inference rule as an axiom, we do so. We therefore only allow the two inference rules that we will need and that it turns out that we cannot encode as axioms: MP and UG.

[3]The string `'A'` in this tuple stands for "Assumption/Axiom," and has nothing to do with the universal quantifier ∀, even though we use the same character to represent both in Python.

The constructor of class `Proof` accepts as its parameters the conclusion, the assumptions/axioms, and the lines of the proof. As in Exercise 4, the main logic of the `Proof` class is in checking the validity of a proof. The method `is_valid()` of this class is implemented for you, however it is missing several key components, which deal with verification of each of the four possible justification types a line can have. In the final tasks of this exercise, you will implement these components.

Task 5. Implement the missing code for the method `verify_a_justification(self, line)` of class `Proof`, which takes an index of a line in the current proof whose justification is declared as `'A'` (see case 1 above), and returns whether this line is validly justified (i.e., whether it really is an instantiation of an assumption/axiom).

We move on to checking whether a line is a tautology. First-order formulae generalize propositional formulae by replacing the propositional atoms with structured subformulae whose root is a relation, an equality, or a quantifier. We define the **propositional skeleton** of a first-order formula as the propositional formula that consistently replaces each of these subformulae with a new name of a propositional atom. For example, the propositional skeleton of '(R(x) | Q(y))→R(x))' is '((z1 | z2)→z1)', the propositional skeleton of '(~x=s(0)→GT(x,1))' is '(~z1→z2)', and the propositional skeleton of '∀x[(R(x)→R(x))]' is 'z1'. We define that a formula in first-order logic is a **tautology** if and only if its propositional skeleton (a propositional formula) is a tautology.

Task 6. Implement the missing code for the method `propositional_skeleton(self)` of class `Formula` (in the file `predicates/syntax.py`), which returns the propositional skeleton of the current first-order formula. The returned object should be of type `propositions.syntax.Formula` (to avoid naming conflicts, this class is imported for you into `predicates/syntax.py` under the name `PropositionalFormula`). The atomic propositional formulae in the returned propositional formula should be named z1,z2,..., ordered according to their first (left-most) occurrence in the original first-order formula, with the numbering increasing between successive calls to `propositional_skeleton`. Call `next(fresh_variable_name_generator)` (this generator is imported for you from `predicates/util.py`) to generate these variable names.

Task 7. Implement the missing code for the method `verify_t_justification(self, line)` of class `Proof`, which takes an index of a line in the current proof whose justification is declared as `'T'` (see case 2 above), and returns whether this line is validly justified (i.e., whether it really is an tautology).[4]

Task 8. Implement the missing code for the method `verify_mp_justification(self, line)` of class `Proof`, which takes an index of a line in the current proof whose justification is declared as `'MP'` (see case 3 above), and returns whether this line is validly justified (i.e., whether it really results from a valid application of *Modus Ponens* to previous lines).

Task 9. Implement the missing code for the method `verify_ug_justification(self, line)` of class `Proof`, which takes an index of a line in the current proof whose justification is declared as `'UG'` (see case 4 above), and returns whether this line is validly justified (i.e., whether it really results from a valid application of Universal Generalization to a previous line).

[4] As mentioned above, we allow tautology justifications purely for simplicity. By inlining the proofs of all needed tautologies we could have done away with these justifications, as well as with their *semantic* validation, resulting in purely syntactic validation of proofs, as one may desire.