This material will be published by Cambridge University Press as:
Mathematical Logic through Python by **Yannai A. Gonczarowski and Noam Nisan**
This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale or use in derivative works. **Please link to: www.LogicThruPython.org**
© Yannai A. Gonczarowski and Noam Nisan 2017–2020.

Chapter 9:

Deductive Proofs of Predicate Logic Formulas

In this chapter, we will develop the notion of formal deductive proofs for Predicate 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 Predicate Logic. The effort that we made in the study of Propositional Logic in the first part of this book will bare its fruits as it will allow us to use arbitrary tautologies as elements of our proofs.

This chapter is rather technical, focusing on the exact syntactic details of our proof system for Predicate Logic. Before we dive into the details, let us give a high level overview of the components of our proof system:

1. **Assumptions and Axioms:** A line in a proof may be an assumption or a logical axiom, which we will treat in the same way. The specific set of axioms that we will use will only be specified in the next chapter. The main complication here is that we allow syntactic *families* of formulas, called **schemas**, as axioms. For example, we would like to be able to have a single axiom schema that says that for any formula ϕ, any variable name x, and any term τ, the following is an axiom: '$(\forall x[\phi(x)]\rightarrow\phi(\tau))$'.

2. **Tautologies:** We will allow using any tautology of Propositional Logic as a line in our proof (we will see below precisely how a tautology of Propositional Logic can be used in a predicate-logic proof). We allow ourselves this power as we already know from our study of Propositional Logic that any such tautology has a proof from some short list of propositional-logic axioms.

3. **Inference Rules:** We will have exactly two inference rules. The first is our old acquaintance Modus Ponens (MP) that deduces the formula 'ψ' from previous lines 'ϕ' and '$(\phi\rightarrow\psi)$'. The second is called Universal Generalization and allows arbitrary universal quantifications: deducing a formula '$\forall x[\phi]$' from a previous line 'ϕ'.

The structure of this chapter is as follows: we start by handling the notion of schemas. We continue by fully specifying our proof system and making sure that it is *sound*: anything that is syntactically proven from some assumptions indeed semantically follows from them. Finally, we revisit our choice of allowing arbitrary tautologies in our proof system and show that by your solutions to the tasks of Chapter 6, this really is only a matter of convenience.

1 Schemas

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 a specialization of this axiom that we are also allowed to use. 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. There are various possible levels of generality for defining schemas, and, as usual, we choose some sufficient intermediate level of generality, trading-off elegance, ease of implementation, and ease of use.

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, here 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' are **templates** that could be (consistently) replaced by any terms but 'plus' cannot be replaced by any other function. In contrast, we may wish to write an assumption such as 'plus(0,c)=c' where 'c' is a template that could be replaced by any term but neither 'plus' nor '0' can be replaced. As a different example, we will also wish to represent some **axiom schema** that stands for a *collection* of axioms of a certain type. That is, in traditional mathematical textual proofs we can say things like "for any formula ϕ, any term τ, and any variable name 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 relation names as templates for formulas, constant names as templates for terms, and variable names as templates for variable names, so we will represent this axiom schema as '(∀x[R(x)]→R(c))' while explicitly stating that 'R', 'x', and 'c' are all templates. In this case, roughly speaking, 'R' is template for any formula into which "a term can be substituted," 'c' is a template for any term, and 'x' is a template for any variable name, and any choice of such formula, term, and variable name to substitute for the templates creates a different **instance** of this axiom schema. All constant, variable, and relation names (if any) that we do not specify to be templates should always be taken literally in any instance of the axiom schema.

1.1 Substitutions

Before we get into the exact details of schemas,[1] we start by implementing an important building block for defining schemas: the notion of *substitution* in a term or formula, where all occurrences of a constant name or variable name are "replaced" by some term. For example, If we have a formula 'c=c', we may substitute the term 'f(x,0)' for the constant name 'c' to get the formula 'f(x,0)=f(x,0)'. Intuitively, if 'c' is only a "placeholder" then the latter formula is a "special case" of the former. One does have to be careful, however, that no variable in the replacing term becomes unintentionally bound in the formula. Consider for example the formula '∃x[f(x,0)=c]' where 'c' is a placeholder. While, for example, it is "fair game" to replace 'c' with the term 'g(y)' to get '∃x[f(x,0)=g(y)]', substituting 'g(x)' into 'c' to get '∃x[f(x,0)=g(x)]' changes the semantics completely so we will forbid such substitutions.

We will sometimes also wish to replace a variable name with a term in a similar fashion, only now we have to be a bit careful to only replace *free* occurrences of the variable name. Consider for example the formula '(R(x)&∀x[Q(x)])'. If we are asked to replace the placeholder 'x' in this formula with the term 'c', it is quite clear that only the

[1]Another frequently used plural form of "schemas," which you may encounter in many books, is "schemata." For simplicity, in this book we will stick with "schemas."

'x' in 'R(x)' should be replaced, since the 'x' in 'Q(x)' refers to the variable universally quantified over ('∀x') inside the part half of the formula.[2]

In the following two tasks you will implement the mechanics of the above-described substitution as part of the classes `Term` and `Formula` in the file `predicates/syntax.py`.

Task 1. Implement the missing code for the method `substitute(substitution_map, forbidden_variables)` of class `Term`, which returns the term obtained from the current term by replacing each occurrence of each variable or constant name that is a key of the given substitution map with the term to which it is mapped. This method raises a `ForbiddenVariableError` (an exception defined in the file `predicates/syntax.py`) if a term that is used in the requested substitution contains one of the given forbidden variable names.

predicates/syntax.py

```
class ForbiddenVariableError(Exception):
    """Raised by `Term.substitute` and `Formula.substitute` when a substituted
    term contains a variable name that is forbidden in that context.

    Attributes:
        variable_name: the variable name that was forbidden in the context in
            which a term containing it was to be substituted.
    """
    variable_name: str

    def __init__(self, variable_name: str):
        """Initializes a `ForbiddenVariableError` from the offending variable
        name.

        Parameters:
            variable_name: variable name that is forbidden in the context in
                which a term containing it is to be substituted.
        """
        assert is_variable(variable_name)
        self.variable_name = variable_name
  ⋮
class Term:
        ⋮
    def substitute(self, substitution_map: Mapping[str, Term],
                   forbidden_variables: AbstractSet[str] = frozenset()) -> Term:
        """Substitutes in the current term, each constant name `name` or
        variable name `name` that is a key in `substitution_map` with the term
        `substitution_map[name]`.

        Parameters:
            substitution_map: mapping defining the substitutions to be
                performed.
            forbidden_variables: variables not allowed in substitution terms.

        Returns:
            The term resulting from performing all substitutions. Only
            constant names and variable names originating in the current term
```

[2]Later in this chapter we will also consider a different situation where we would want to replace a variable name such as 'x' with another *variable name*, say, 'z', allowing us to replace every occurrence of the former, including the quantification itself, in this case giving '(R(z)&∀z[Q(z)])', but for now we stick with the basic semantics of replacing only unbound variable occurrences with arbitrary terms.

```
            are substituted (i.e., those originating in one of the specified
            substitutions are not subjected to additional substitutions).

        Raises:
            ForbiddenVariableError: If a term that is used in the requested
                substitution contains a variable from `forbidden_variables`.

        Examples:
            >>> Term.parse('f(x,c)').substitute(
            ...     {'c': Term.parse('plus(d,x)'), 'x': Term.parse('c')}, {'y'})
            f(c,plus(d,x))

            >>> Term.parse('f(x,c)').substitute(
            ...     {'c': Term.parse('plus(d,y)')}, {'y'})
            Traceback (most recent call last):
              ...
            predicates.syntax.ForbiddenVariableError: y
        """
        for element_name in substitution_map:
            assert is_constant(element_name) or is_variable(element_name)
        for variable in forbidden_variables:
            assert is_variable(variable)
        # Task 9.1
```

Hint: Use recursion.

Task 2. Implement the missing code for the method `substitute(substitution_map, forbidden_variables)` of class `Formula`, which returns the formula obtained from the current formula by replacing each occurrence of each constant name that is a key of the given substitution map with the term to which it is mapped, and replacing each *free* occurrence of each variable name that is a key of the given substitution map with the term to which it is mapped. This method raises a `ForbiddenVariableError` if a term that is used in the requested substitution contains a variable name that either is one of the given forbidden variable names or becomes bound when the term is substituted into the formula.

predicates/syntax.py

```
class Formula:
        ⋮
    def substitute(self, substitution_map: Mapping[str, Term],
                   forbidden_variables: AbstractSet[str] = frozenset()) -> \
                Formula:
        """Substitutes in the current formula, each constant name `name` or free
        occurrence of variable name `name` that is a key in `substitution_map`
        with the term `substitution_map[name]`.

        Parameters:
            substitution_map: mapping defining the substitutions to be
                performed.
            forbidden_variables: variables not allowed in substitution terms.

        Returns:
            The formula resulting from performing all substitutions. Only
            constant names and variable names originating in the current formula
            are substituted (i.e., those originating in one of the specified
            substitutions are not subjected to additional substitutions).
```

```
        Raises:
            ForbiddenVariableError: If a term that is used in the requested
                substitution contains a variable from `forbidden_variables`
                or a variable occurrence that becomes bound when that term is
                substituted into the current formula.

        Examples:
            >>> Formula.parse('Ay[x=c]').substitute(
            ...     {'c': Term.parse('plus(d,x)'), 'x': Term.parse('c')}, {'z'})
            Ay[c=plus(d,x)]

            >>> Formula.parse('Ay[x=c]').substitute(
            ...     {'c': Term.parse('plus(d,z)')}, {'z'})
            Traceback (most recent call last):
              ...
            predicates.syntax.ForbiddenVariableError: z

            >>> Formula.parse('Ay[x=c]').substitute(
            ...     {'c': Term.parse('plus(d,y)')})
            Traceback (most recent call last):
              ...
            predicates.syntax.ForbiddenVariableError: y
        """
        for element_name in substitution_map:
            assert is_constant(element_name) or is_variable(element_name)
        for variable in forbidden_variables:
            assert is_variable(variable)
        # Task 9.2
```

Hint: Use recursion, augmenting the set `forbidden_variables` in the recursive call with quantified variables as needed. In the recursion base use your solution to Task 1.

1.2 Schemas

We can now move on to defining (and programming) exactly what a *schema* is and which formulas are **instances** of it. The file `predicates/proofs.py` defines (among other classes) a Python class `Schema` that represents a schema as a regular predicate-logic formula together with the set of elements of this formula that serve as templates (placeholders).

predicates/proofs.py

```
@frozen
class Schema:
    """An immutable schema of predicate-logic formulas, comprised of a formula
    along with the constant names, variable names, and nullary or unary relation
    names in that formula that serve as templates. A template constant name is a
    placeholder for any term. A template variable name is a placeholder for any
    variable name. A template nullary or unary relation name is a placeholder
    for any (parametrized for a unary relation name) predicate-logic formula
    that does not refer to any variable name in the schema (except possibly
    through its instantiated parameter for a unary relation name).

    Attributes:
        formula: the formula of the schema.
        templates: the constant, variable, and relation names from the formula
```

```
            that serve as templates.
    """
    formula: Formula
    templates: FrozenSet[str]

    def __init__(self, formula: Formula,
                 templates: AbstractSet[str] = frozenset()):
        """Initializes a `Schema` from its formula and template names.

        Parameters:
            formula: the formula for the schema.
            templates: the constant, variable, and relation names from the
                formula to serve as templates.
        """
        for template in templates:
            assert is_constant(template) or is_variable(template) or \
                   is_relation(template)
            if is_relation(template):
                arities = {arity for relation,arity in formula.relations() if
                           relation == template}
                assert arities == {0} or arities == {1}
        self.formula = formula
        self.templates = frozenset(templates)
```

The default value (for the constructor of this class) of the set of templates is the empty set, which corresponds to a schema with only one possible instance (because there are no templates that can be replaced): the original schema formula. Three types of syntactic elements can potentially serve as templates: constant names, variables names, and relation invocations.

Constant Names

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

```
Schema(Formula.parse('c=c'), {'c'})
```

has as specific instances all of the following formulas (among others): '0=0', 'x=x', 'plus(x,y)=plus(x,y)'.

Variable Names

A variable name that is specified as a template can serve as a placeholder for any variable name. (This is a different semantic than in the basic substitution described in Tasks 1 and 2!) For example, the schema

```
Schema(Formula.parse('(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 immediately follow a quantification.

Relation Names

A relation that is specified as a template can serve as a placeholder for an arbitrary formula, that may possibly be "parametrized" by a single "parameter." Let us first consider the simpler case of a **parameter-less** relation template—a template whose invocations in the schema formula are nullary, such as is the case in the schema

```
Schema(Formula.parse('(Q()|~Q())'), {'Q'})
```

for the template 'Q'. A parameter-less relation template such as 'Q' in this schema can serve as a placeholder for any formula. Thus, for example, this schema has the following as in instance: '(c=y|~c=y)'.

Let us now consider the more intricate case of a **parametrized** relation template—a template whose invocations in the schema formula are unary, such as is the case in the schema

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

for each of the templates 'Q' and 'R'. (The constructor of class `Schema` verifies for each template relation name that its invocations in the schema formula are either all nullary—corresponding to a parameter-less template—or all unary—corresponding to a parametrized template). A parametrized relation template such as 'Q' or 'R' in this schema can serve as a placeholder for any **parametrized formula**, that is, a formula that optionally contains a placeholder for the argument of the unary invocation. Thus, for example, this schema can be **instantiated** with 'R(□)' defined as '□=7' (parametrized the placeholder □), and with 'Q(□)' defined as 'T(y,□)' to obtain the following instance of this schema: '(∀x[(x=7→T(y,x))]→(∀z[z=7]→∀w[T(y,w)]))'.

Note that when defining the parametrized template 'Q(□)' as 'T(y,□)' in the last example, and when defining the parameter-less template 'Q()' as 'x=y' in the preceding example, we allowed the formula that replaces the relation template to contain unbound variables names ('y' in both examples). In order to avoid unintended (and logically incorrect) quantifications, we however restrict this to only be allowed for unbound variable names that *do not get bound by a quantifier in the resulting instance of the schema.* Thus, for example, if we look at the schema

```
Schema(Formula.parse('(Ax[R(x)]->R(c))'), {'R', 'c', 'x'})
```

given in the first example above, then we allow to instantiate this schema with 'R(□)' defined as '□=0' to get '(∀x[x=0]→c=0)', but we do not allow to instantiate it with 'R(□)' defined as '□=x', so '(∀x[x=x]→c=x)' is *not* an instance of this schema (nor is it a logically valid statement), since the unbound variable name 'x' in '□=x' would get bound by the universal quantifier in the resulting schema instance if such a substitution were to be made.

Let us look more closely at the process of, e.g., taking the above schema

```
Schema(Formula.parse('(Ax[R(x)]->R(c))'), {'R', 'c', 'x'})
```

and instantiating it with 's(1)' for 'c', with '(∃z[s(z)=□]→Q(□))' for 'R(□)', and with 'y' for 'x':

1. The variable name 'x' that immediately follows the universal quantification gets substituted for 'y'.

DRAFT

2. For the first occurrence of 'R' in the schema (i.e., for 'R(x)') we first substitute 'y' for 'x' to get the **instantiated argument** 'y' of that invocation of this template relation. This instantiated argument then gets substituted for all occurrences of the parameter '□' in the parametrized formula '(∃z[s(z)=□]→Q(□))' to get '(∃z[s(z)=y]→Q(y))' as the final replacement for 'R(x)'.

3. For the second occurrence (i.e., for 'R(c)') we substitute 's(1)' for 'c' to get the instantiated argument 's(1)' of that invocation of this template relation. This instantiated argument then gets substituted for all occurrences of the parameter '□' in the parametrized formula '(∃z[s(z)=□]→Q(□))' to get '(∃z[s(z)=s(1)]→Q(s(1)))' as the final replacement for 'R(c)'.

4. Altogether, we get '(∀y[(∃z[s(z)=y]→Q(y))]→(∃z[s(z)=s(1)]→Q(s(1))))' as the resulting instance of this axiom schema (which is indeed a logically valid statement).

One final restriction we must verify regards quantification *within* parametrized formulas that are substituted for parametrized template relations. Consider again the schema

```
Schema(Formula.parse('(Ax[R(x)]->R(c))'), {'R', 'c', 'x'})
```

as above. If we were to allow to instantiate this schema with 'R(□)' as '∃x[□=7]', then we would get '(∀x[∃x[x=7]]→∃x[c=7])', which is logically invalid. Therefore, in parametrized formulas that replace (parametrized) relation templates, we do not allow quantifications that *bound any variable name in any instantiated argument of that parametrized relation.* Note that the method `substitute` of class `Formula` that you implemented in Task 2 indeed would raise an exception when substituting the instantiated argument into the parametrized formula in such disallowed cases (and would not raise one in allowed cases).[3]

All Together Now

We can now put all of these substitutions together and implement the most important functionality of the `Schema` class, which is to *instantiate* the schema according to a given **instantiation map**. We will represent parametrized formulas in Python as regular `Formula` objects that use the constant name '_' (underscore) as the placeholder. So, for example, the first parametrized formula given as an example above, '□=7', would be

[3] Some readers may at this point wonder why we do not disallow some other "strange instantiations." Consider, e.g., the "intuitively correct" schema `Schema(Formula.parse('Ex[R(c)]->R(c)'), {'c'})`. Replacing 'c' with 'x' in this schema would yield the logically incorrect '∃x[R(x)]→R(x)', so to avoid that we could have restricted *terms* that replace template *constant* names to contain only unbound variable names that *do not get bound by a quantifier in the resulting instance of the schema*, analogously to the first constraint above regarding variables names in *formulas* that replace template *relation* names. As another example, consider the "intuitively correct" schema `Schema(Formula.parse('Ey[~y=x]'), {'y'})`. Replacing 'y' with 'x' in this schema would yield the logically incorrect '∃x[~x=x]', so to avoid that we could have restricted substitutions of template *variable* names to only those substitutions that *do not make other existing variable names bound by quantifications over the substituted variable names.* While restrictions such as these could have been useful, and would certainly have prevented some instances of some schemas from being logically incorrect, this is in fact not the case for any of the schemas that we will use in this book. More generally, there is no "right" or "wrong" set of restrictions to impose here: all that we need, as we will see in the following chapters, is to have enough restrictions in place on the possible instances of schemas to make sure that the specific schemas that we define and use throughout this book do not have logically incorrect instances (but do still have as legal instances all of the intended logically correct instances that we will need in our proofs). As we will see in Chapter 10, for the schemas that we chose for this book, the two restrictions detailed above in the text suffice.

represented in Python by the formula that is returned by `Formula.parse('_=7')`. To avoid confusion, from this point onward, throughout the book and throughout all tests, we will use the constant name '_' only for this purpose.

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, as well as of substituting an instantiated argument into a parametrized formula that replaced a parametrized template relation. We still need to focus on handling the overall instantiation of parameter-less and parametrized relation templates, verifying that we adhere to the two restrictions above, and correctly combining the instantiation of relation templates with that of instantiation of constant and variable templates. The recursive 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_helper` of class `Schema`. This recursive method takes four arguments:

1. A `formula`.
2. A map `constants_and_variables_instantiation_map` that maps each template constant name and template variable name to a term that is to be substituted for that template in `formula`. Constant names may be mapped to any term, while variable names may only be mapped to a term whose root is a variable name.
3. A map `relations_instantiation_map` that maps each template relation name to a formula that is to be substituted for. For parametrized relation templates, the mapped formula is parametrized by the constant name `'_'`.
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 name in `formula` may not be mapped by `relation_instantiation_map` to a formula that has free variables that are in this set, just like (see below) it may not be mapped to a formula that has free variables that get quantified when plugged into `formula`.

This method returns a formula resulting from `formula` by performing all of the above-described substitutions.

This method raises a `Schema.BoundVariableError` (an exception defined for you in the class `Schema`) if any free variable in any formula from `relations_instantiation_map` that is substituted for a (parameter-less or parametrized) relation template gets bound in the returned formula or is in the set `bound_variables`.

The method also raises a `Schema.BoundVariableError` (this is our recommendation, but for the convenience of readers who are unfamiliar with the details of exception handling, the tests that we provide also accept raising any other exception) if for an invocation of a parametrized relation template, the substitution of the instantiated argument into the parametrized formula to which the template relation is mapped causes a variable in the instantiated argument to get bound by a quantification in that parametrized formula (that is, if the corresponding call to the method `substitute()` of that formula raises an exception).

predicates/proofs.py

```
class Schema:
        ⋮
```

```
class BoundVariableError(Exception):
    """Raised by `_instantiate_helper` when a variable name becomes bound
    during a schema instantiation in a way that is disallowed in that
    context.

    Attributes:
        variable_name: the variable name that became bound in a way that was
            disallowed during a schema instantiation.
        relation_name: the relation name during whose substitution the
            relevant occurrence of the variable name became bound.
    """
    variable_name: str
    relation_name: str

    def __init__(self, variable_name: str, relation_name: str):
        """Initializes a `~Schema.BoundVariableError` from the offending
        variable name and the relation name during whose substitution the
        error occurred.

        Parameters:
            variable_name: variable name that is to become bound in a way
                that is disallowed during a schema instantiation.
            relation_name: the relation name during whose substitution the
                relevant occurrence of the variable name is to become bound.
        """
        assert is_variable(variable_name)
        assert is_relation(relation_name)
        self.variable_name = variable_name
        self.relation_name = relation_name

@staticmethod
def _instantiate_helper(formula: Formula,
                        constants_and_variables_instantiation_map:
                        Mapping[str, Term],
                        relations_instantiation_map: Mapping[str, Formula],
                        bound_variables: AbstractSet[str] = frozenset()) \
        -> Formula:
    """Performs the following substitutions in the given formula:

    1. Substitute each occurrence of each constant name or variable name
       that is a key of the given constants and variables instantiation map
       with the term mapped to this name by this map.
    2. Substitute each nullary invocation of each relation name that is a
       key of the given relations instantiation map with the formula mapped
       to it by this map.
    3. For each unary invocation of each relation name that is a key of the
       given relations instantiation map, first perform all substitutions
       to the argument of this invocation (according to the given constants
       and variables instantiation map), then substitute the result for
       each occurrence of the constant name '_' in the formula mapped to the
       relation name by this map, and then substitute the result for this
       unary invocation of the relation name.

    Only names originating in the given formula are substituted (i.e., names
    originating in one of the above substitutions are not subjected to
    additional substitutions).

    Parameters:
```

```
        formula: formula in which to perform the substitutions.
        constants_and_variables_instantiation_map: mapping from constant
            names and variable names in the given formula to terms to be
            substituted for them, where the roots of terms mapped to
            variable names are variable names.
        relations_instantiation_map: mapping from nullary and unary relation
            names in the given formula to formulas to be substituted for
            them, where formulas to be substituted for unary relation names
            are parametrized by the constant name '_'.
        bound_variables: variables to be treated as bound (see below).

    Returns:
        The result of all substitutions.

    Raises:
        BoundVariableError: if one of the following occurs when substituting
            an invocation of a relation name:

            1. A free occurrence of a variable name in the formula
               mapped to the relation name by the given relations
               instantiation map is in `bound_variables` or becomes bound
               by a quantification in the given formula after all variable
               names in the given formula have been substituted.
            2. For a unary invocation: a variable name that is in the
               argument to that invocation after all substitutions have been
               applied to this argument, becomes bound by a quantification
               in the formula mapped to the relation name by the given
               relations instantiation map.

    Examples:
        The following succeeds:

        >>> Schema._instantiate_helper(
        ...     Formula.parse('Ax[(Q(c)->R(x))]'), {'x': Term('w')},
        ...     {'Q': Formula.parse('y=_')}, {'x', 'z'})
        Aw[(y=c->R(w))]

        however the following fails since 'Q(c)' is to be substituted with
        'y=c' while 'y' is in the given bound variables:

        >>> Schema._instantiate_helper(
        ...     Formula.parse('Ax[(Q(c)->R(x))]'), {},
        ...     {'Q': Formula.parse('y=_')}, {'x', 'y', 'z'})
        Traceback (most recent call last):
          ...
        predicates.proofs.Schema.BoundVariableError: ('y', 'Q')

        and the following fails since as 'Q(c)' is to be substituted with
        'y=c', 'y' is to become bound by the quantification 'Ay':

        >>> Schema._instantiate_helper(
        ...     Formula.parse('Ax[(Q(c)->R(x))]'), {'x': Term('y')},
        ...     {'Q': Formula.parse('y=_')})
        Traceback (most recent call last):
          ...
        predicates.proofs.Schema.BoundVariableError: ('y', 'Q')

        The following succeeds:
```

```
        >>> Schema._instantiate_helper(
        ...     Formula.parse('Ax[(Q(c)->R(x))]'),
        ...     {'c': Term.parse('plus(d,x)')},
        ...     {'Q': Formula.parse('Ey[y=_]')})
        Ax[(Ey[y=plus(d,x)]->R(x))]

        however the following fails since as '_' is to be substituted with
        'plus(d,y)' in 'Ey[y=_]', the 'y' in 'plus(d,y)' is to become bound
        by the quantification 'Ey':

        >>> Schema._instantiate_helper(
        ...     Formula.parse('Ax[(Q(c)->R(x))]'),
        ...     {'c': Term.parse('plus(d,y)')},
        ...     {'Q': Formula.parse('Ey[y=_]')})
        Traceback (most recent call last):
          ...
        predicates.proofs.Schema.BoundVariableError: ('y', 'Q')
    """
    for name in constants_and_variables_instantiation_map:
        assert is_constant(name) or is_variable(name)
        if is_variable(name):
            assert is_variable(
                constants_and_variables_instantiation_map[name].root)
    for relation in relations_instantiation_map:
        assert is_relation(relation)
    for variable in bound_variables:
        assert is_variable(variable)
    # Task 9.3
```

Guidelines: This method should naturally be implemented recursively. Two simple base cases are when `formula` is an invocation of a relation that is not a template, or an equality. The required substitution here is simply given by the `substitute()` method of class `Formula`. Another simple base case is when `formula` is a nullary invocation of a (parameter-less) template relation. In this case, the formula to which this relation is mapped should simply be returned (but only after checking that the free variables of this formula are disjoint from the set `bound_variables`—otherwise an exception should be raised). The interesting base case is when `formula` is a unary invocation of a (parametrized) template relation, that is, of the form '$R(t)$' where R is a template relation name and t is a term. In this case, you should first use the `substitute()` method of class `Term` to make the substitutions in t to obtain the instantiated argument t' of this invocation, and then you should "plug" the instantiated argument t' into the formula ϕ to which this template relation name is mapped by using the `substitute()` method of class `Formula` with the substitution map `{'_':` t'`}`. (Of course, in this case you should also check before that the free variables of ϕ 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 where `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.
Hint: Make sure that you understand why in each of the calls to any of the `substitute()`

methods that are detailed in the above guidelines, an empty set should always be specified as the set of forbidden variables.

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

Task 4. Implement the missing code for the method `instantiate(instantiation_map)` of class `Schema`, which takes an instantiation map—a map that maps each template of the current schema to what it should be instantiated to—and returns the instantiated schema instance, as explained above (between Task 2 and Task 3). Templates that are not mapped by the given instantiation map remain as is in the returned instance. If the instantiation map specifies a constant, variable, or relation name that is not a template, or if an illegal instantiation (of one of the two types explained above) occurs, then `None` is returned instead.

predicates/proofs.py

```
#: A mapping from constant names, variable names, and relation names to
#: terms, variable names, and formulas respectively.
InstantiationMap = Mapping[str, Union[Term, str, Formula]]
⋮
class Schema:
        ⋮
    def instantiate(self, instantiation_map: InstantiationMap) -> \
            Union[Formula, None]:
        """Instantiates the current schema according to the given map from
        templates of the current schema to expressions.

        Parameters:
            instantiation_map: mapping from templates of the current schema to
                expressions of the type for which they serve as placeholders.
                That is, constant names are mapped to terms, variable names are
                mapped to variable names (strings), and relation names are
                mapped to formulas where unary relations are mapped to formulas
                parametrized by the constant name ``'_'``.

        Returns:
            The predicate-logic formula obtained by applying the substitutions
            specified by the given map to the formula of the current schema:

            1. Each occurrence in the formula of the current schema of each
               template constant name specified in the given map is substituted
               with the term to which that template constant name is mapped.
            2. Each occurrence in the formula of the current schema of each
               template variable name specified in the given map is substituted
               with the variable name to which that template variable name is
               mapped.
            3. Each nullary invocation in the formula of the current schema of
               each template relation name specified in the given map is
               substituted with the formula to which that template relation name
               is mapped.
            4. Each unary invocation in the formula of the current schema of
               each template relation name specified in the given map is
               substituted with the formula to which that template relation name
               is mapped, in which each occurrence of the constant name '_' is
               substituted with  the instantiated argument of that invocation of
               the template relation name (that is, the term that results from
               instantiating the argument of that invocation by performing all
```

```
        the specified substitutions on it).

    ``None`` is returned if one of the keys of the given map is not a
    template of the current schema or if one of the following occurs
    when substituting an invocation of a template relation name:

    1. A free occurrence of a variable name in the formula substituted
       for the template relation name becomes bound by a quantification
       in the instantiated schema formula, except if the template
       relation name is unary and this free occurrence originates in the
       instantiated argument of the relation invocation.
    2. For a unary invocation: a variable name in the instantiated
       argument of that invocation becomes bound by a quantification in
       the formula that is substituted for the invocation of the
       template relation name.

Examples:
    >>> s = Schema(Formula.parse('(Q(c1,c2)->(R(c1)->R(c2)))'),
    ...            {'c1', 'c2', 'R'})
    >>> s.instantiate({'c1': Term.parse('plus(x,1)'),
    ...                'R': Formula.parse('Q(_,y)')})
    (Q(plus(x,1),c2)->(Q(plus(x,1),y)->Q(c2,y)))
    >>> s.instantiate({'c1': Term.parse('plus(x,1)'),
    ...                'c2': Term.parse('c1'),
    ...                'R': Formula.parse('Q(_,y)')})
    (Q(plus(x,1),c1)->(Q(plus(x,1),y)->Q(c1,y)))

    >>> s = Schema(Formula.parse('(P()->P())'), {'P'})
    >>> s.instantiate({'P': Formula.parse('plus(a,b)=c')})
    (plus(a,b)=c->plus(a,b)=c)

    For the following schema:

    >>> s = Schema(Formula.parse('(Q(d)->Ax[(R(x)->Q(f(c)))])'),
    ...            {'R', 'Q', 'x', 'c'})

    the following succeeds:

    >>> s.instantiate({'R': Formula.parse('_=0'),
    ...                'Q': Formula.parse('x=_'),
    ...                'x': 'w'})
    (x=d->Aw[(w=0->x=f(c))])

    however, the following returns ``None`` because 'd' is not a
    template of the schema:

    >>> s.instantiate({'R': Formula.parse('_=0'),
    ...                'Q': Formula.parse('x=_'),
    ...                'x': 'w',
    ...                'd': Term('z')})

    and the following returns ``None`` because 'z' that is free in the
    assignment to 'Q' is to become bound by a quantification in the
    instantiated schema formula:

    >>> s.instantiate({'R': Formula.parse('_=0'),
    ...                'Q': Formula.parse('s(z)=_'),
    ...                'x': 'z'})
```

```
        and the following returns ``None`` because 'y' in the instantiated
        argument 'f(plus(a,y))' of the second invocation of 'Q' is to become
         bound by the quantification in the formula substituted for 'Q':

        >>> s.instantiate({'R': Formula.parse('_=0'),
        ...                'Q': Formula.parse('Ay[s(y)=_]'),
        ...                'c': Term.parse('plus(a,y)')})
    """
    for key in instantiation_map:
        if is_variable(key):
            assert is_variable(instantiation_map[key])
        elif is_constant(key):
            assert isinstance(instantiation_map[key], Term)
        else:
            assert is_relation(key)
            assert isinstance(instantiation_map[key], Formula)
    # Task 9.4
```

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 exceptions raised by `_instantiate_helper()`.

We conclude this section by recalling the Specialization Soundness Lemma for Propositional Logic, which states that every specialization of a sound inference rule is itself sound as well. While it would have been nice to have an analogous lemma for Predicate Logic, it is not clear exactly what such a lemma would mean: that every instance of a sound schema is sound? But what does it mean for a schema to be sound? In Propositional Logic since both an inference rule and its specializations are inference rules, this lemma has nontrivial meaning since there is a definition, that does not involve specialization, for what it means for an inference rule to be sound. In Predicate Logic, however, formulas are instances of schemas rather than specializations of formulas, and while it is clear what it means for a formula to be sound—to hold in every model for that formula—it is not clear what it means for a schema to be sound (without mentioning instantiation). Indeed, what would it mean for a schema to hold for every model for that schema? Therefore, in Predicate Logic, we do not have an analogous lemma, but rather we define the soundness property of schemas so that this statement holds by definition.

Definition (Sound Formula; Sound Schema)**.** We say that a predicate-logic formula is **sound** if it holds in every model for it, for any assignment to its free variables. We say that a schema is **sound** if every instance of it (an instance is a predicate-logic formula) is sound.

In fact, as we will see in the next chapter, the two restrictions on instantiations that we imposed above are in place to make sure that our standard axiom schemas—see the next chapter—are sound: to disallow "instances" of these schemas that would not hold in all models, and would thus render these schemas not sound.

Before moving on to defining proofs in Predicate Logic, we note an important difference in the definition of soundness between Propositional Logic and Predicate Logic. Unlike in Propositional Logic, where the soundness of any inference rule (or formula) could be checked using a finite semantic procedure (checking all of the finitely many possible models by calling `is_sound_inference()`), in Predicate Logic there may be

DRAFT

infinitely many models for a given formula, and so just semantically checking soundness by going over all possible models is infeasible.[4] We will discuss this point in the next chapter. For the next section, we will assume that we are somehow assured that some basic schemas or formulas are sound, and we will wish to use them to prove that other formulas are sound.

2 Proofs

We can now move to defining formal deductive proofs in Predicate Logic. Just like in Propositional 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. Of course, here we will have different axioms and inference rules, and allow different assumptions, than in Propositional Logic. There are many possible variants for the allowed inference rules and logical axioms for Predicate Logic. We will use a system that has two inference rules (that have assumptions) rather than one in Propositional Logic,[5] and a handful of axiom schemas that we will get to know in the next chapter. More specifically, we will allow the following types of justifications for the lines in our proofs:

1. **Assumption/Axiom**: We may, of course, use any of our assumptions/axioms of the proof in it, and note that as our assumptions/axioms are given as schemas, any instance of an assumed/axiomatic schema may be used. For simplicity, we do not make any programmatic distinction between assumptions and axioms, however one may think of assumptions/axioms that are schemas 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. **Modus Ponens**: We will keep allowing Modus Ponens, or MP, as an inference rule. That is, from ϕ and '$(\phi \rightarrow \psi)$' (that are conclusions of previous proof lines) we may deduce ψ.

3. **Universal Generalization**: We introduce one new allowed inference rule, named Universal Generalization, or UG. That is, from any formula ϕ (that is a conclusion of a previous proof line), for any variable x we may deduce '$\forall x[\phi]$'. For example, from the formula '(R(x)→Q(x))' we may deduce '∀x[(R(x)→Q(x))]' as well as '∀z[(R(x)→Q(x))]'. As we will see below, the UG rule syntactically encompasses our *semantic* treatment of free variables as being universally quantified.

4. **Tautology**: Finally, we will also allow the use of any **predicate-logic tautology** without proof, where a tautology is a formula that is true when viewed as a propo-

[4] A certain set of formulas for which there is a finite semantic procedure for verifying their soundness—**tautologies** (in Predicate Logic this is no longer a synonym for a sound formula, but rather a special case of a sound formula)—will be mentioned in Section 2 and discussed in Section 3.

[5] For simplicity, in our predicate-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 have assumptions) that we will need and that it turns out that we cannot encode as axioms: our tried-and-true MP and a newly introduced called UG (see below).

sitional formula (see below for the precise definition). For example, '(R(x)→R(x))' is a tautology, and so is '(∀x[R(x)]→∀x[R(x)])', but '∀x[(R(x)→R(x))]' is not a tautology (once again, see below for the precise definition). While attentive readers would at this point protest the blatant usage within proofs of something that has to be semantically (albeit finitely) checked (i.e., whether the given formula is a tautology), we note that allowing any tautology is purely for simplicity, as we could have alternatively added a small set of schemas, each representing one of the our axioms from Chapter 6 as assumptions/axioms, and "inlined" the proof of any needed tautology using these axioms. We show how to do this in Section 3 below.

For example, here is a simple proof that uses each of the four justification types defined above:

Assumption: 'R(c)', where c is a template.

Conclusion: '∀x[~~R(x)]'

Proof:

1. 'R(x)'. Justification: instance of the assumption, instantiated with c defined as 'x'.
2. '(R(x)→~~R(x))'. Justification: a tautology.
3. '~~R(x)'. Justification: MP from from Steps 1 and 2.
4. '∀x[~~R(x)]'. Justification: UG of Step 3.

The file `predicates/proofs.py` defines the Python class `Proof` that represents such a proof, and contains a set of assumptions/axioms, each of them a `Schema`, a conclusion that is a `Formula` object, and the body of the proof that consists of its lines (see below).

predicates/proofs.py

```
@frozen
class Proof:
    """An immutable deductive proof in Predicate Logic, comprised of a list of
    assumptions/axioms, a conclusion, and a list of lines that prove the
    conclusion from (instances of) these assumptions/axioms and from
    tautologies, via the Modus Ponens (MP) and Universal Generalization (UG)
    inference rules.

    Attributes:
        assumptions: the assumption/axioms of the proof.
        conclusion: the conclusion of the proof.
        lines: the lines of the proof.
    """
    assumptions: FrozenSet[Schema]
    conclusion: Formula
    lines: Tuple[Proof.Line, ...]

    def __init__(self, assumptions: AbstractSet[Schema], conclusion: Formula,
                 lines: Sequence[Proof.Line]):
        """Initializes a `Proof` from its assumptions/axioms, conclusion,
        and lines.

        Parameters:
            assumptions: the assumption/axioms for the proof.
            conclusion: the conclusion for the proof.
```

```
            lines: the lines for the proof.
        """
        self.assumptions = frozenset(assumptions)
        self.conclusion = conclusion
        self.lines = tuple(lines)
```

Unlike in Propositional Logic where we had one class that represented any line with one of the two allowed types of line justifications (being an assumption or being the conclusion of an allowed inference rule), here we will have a different class for each of the four allowed types of line justifications. Each of these four line classes will adhere to the following "interface":

- It is immutable.
- It has a field called `formula` that contains the formula justified by the line.
- It has a method `is_valid(assumptions, lines, line_number)` that checks if the line validly justifies its conclusion given the assumptions/axioms of the proof and given the other lines of the proof (the method is also given the line number of the current line within the lines of the proof).[6]

This unified "interface" allows code that operates on the lines of a proof to handle all lines similarly and transparently. For example, this allows the following simple implementation of the method `is_valid()` of class `Proof` that is implemented for you, which checks the validity of the current proof:

predicates/proofs.py

```
class Proof:
        ⋮
    #: An immutable proof line.
    Line = Union[AssumptionLine, MPLine, UGLine, TautologyLine]
    ⋮
    def is_valid(self) -> bool:
        """Checks if the current proof is a valid proof of its claimed
        conclusion from (instances of) its assumptions/axioms.

        Returns:
            ``True`` if the current proof is a valid proof of its claimed
            conclusion from (instances of) its assumptions/axioms, ``False``
            otherwise.
        """
        if len(self.lines) == 0 or self.lines[-1].formula != self.conclusion:
            return False
        for line_number in range(len(self.lines)):
            if not self.lines[line_number].is_valid(self.assumptions,
                                                    self.lines, line_number):
                return False
        return True
```

[6]The motivation for the design decision of having this method take the assumptions of the proof and the lines of the proof as two separate arguments rather than just take the entire `Proof` as one argument will become clear in the next chapter.

As in Chapter 4, the main functional aspect of the `Proof` class is in checking the validity of a proof. While the method `is_valid()` of this class is implemented for you, it is missing its core components that deal with verification of the four allowed types of justification types—the implementations for the `is_valid()` methods of the various classes of proof lines. In the tasks of this section, you will implement these components.

2.1 Assumption Lines

The class `Proof.AssumptionLine` is used for proof lines justified as instances of assumptions/axioms. This class holds, in addition to the formula that it justifies, also the assumption/axiom (a schema) whose instance is this formula, as well as the instantiation map according to which this assumption/axiom can be instantiated to obtain this formula (the map may be empty if the assumption/axiom has no templates).

predicates/proofs.py

```
class Proof:
        ⋮
    @frozen
    class AssumptionLine:
        """An immutable proof line justified as an instance of an
        assumption/axiom.

        Attributes:
            formula: the formula justified by the line.
            assumption: the assumption/axiom that instantiates the formula.
            instantiation_map: the mapping instantiating the formula from the
                assumption/axiom.
        """
        formula: Formula
        assumption: Schema
        instantiation_map: InstantiationMap

        def __init__(self, formula: Formula, assumption: Schema,
                     instantiation_map: InstantiationMap):
            """Initializes an `Proof.AssumptionLine` from its formula, its
            justifying assumption, and its instantiation map from the justifying
            assumption.

            Parameters:
                formula: the formula to be justified by the line.
                assumption: the assumption/axiom that instantiates the formula.
                instantiation_map: the mapping instantiating the formula from
                    the assumption/axiom.
            """
            self.formula = formula
            self.assumption = assumption
            for key in instantiation_map:
                if is_variable(key):
                    assert is_variable(instantiation_map[key])
                elif is_constant(key):
                    assert isinstance(instantiation_map[key], Term)
                else:
                    assert is_relation(key)
                    assert isinstance(instantiation_map[key], Formula)
            self.instantiation_map = frozendict(instantiation_map)
```

Task 5. Implement the missing code for the method `is_valid(assumptions, lines, line_number)` of class `Proof.AssumptionLine`, which returns whether the formula of the current line is validly justified within the context of the specified proof (i.e., whether it really is an instantiation, as specified, of one of the assumption/axiom of this proof).

predicates/proofs.py

```
class Proof:
        ⋮
    class AssumptionLine:
                  ⋮
        def is_valid(self, assumptions: AbstractSet[Schema],
                     lines: Sequence[Proof.Line], line_number: int) -> bool:
            """Checks if the current line is validly justified in the context of
            the specified proof.

            Parameters:
                assumptions: assumptions/axioms of the proof.
                lines: lines of the proof.
                line_number: line number of the current line in the given lines.

            Returns:
                ``True`` if the assumption/axiom of the current line is an
                assumption/axiom of the specified proof and if the formula
                justified by the current line is a valid instance of this
                assumption/axiom via the instantiation map of the current line,
                ``False`` otherwise.
            """
            assert line_number < len(lines) and lines[line_number] is self
            # Task 9.5
```

Hint: Recall that this method takes the three arguments `assumptions`, `lines`, and `line_number` in order to take the same arguments as the `is_valid()` methods of other proof line classes. You need not use all of these in your solution to this task.

2.2 Modus Ponens (MP) Lines

The class `Proof.MPLine` is used for proof lines justified by the MP inference rule. This class holds, in addition to the formula that it justifies, also the line numbers of the previous lines from which this formula is deduced via MP.

predicates/proofs.py

```
class Proof:
        ⋮
    @frozen
    class MPLine:
        """An immutable proof line justified by the Modus Ponens (MP) inference
        rule.

        Attributes:
            formula: the formula justified by the line.
            antecedent_line_number: the line number of the antecedent of the MP
                inference justifying the line.
            conditional_line_number: the line number of the conditional of the
                MP inference justifying the line.
        """
```

```
        formula: Formula
        antecedent_line_number: int
        conditional_line_number: int

        def __init__(self, formula: Formula, antecedent_line_number: int,
                     conditional_line_number: int):
            """Initializes a `Proof.MPLine` from its formula and line numbers of
            the antecedent and conditional of the MP inference justifying it.

            Parameters:
                formula: the formula to be justified by the line.
                antecedent_line_number: the line number of the antecedent of the
                    MP inference justifying the line.
                conditional_line_number: the line number of the conditional of
                    the MP inference justifying the line.
            """
            self.formula = formula
            self.antecedent_line_number = antecedent_line_number
            self.conditional_line_number = conditional_line_number
```

Task 6. Implement the missing code for the method `is_valid(assumptions, lines, line_number)` of class `Proof.MPLine`, which returns whether the formula of the current line is validly justified via an application of MP to the specified previous lines.

predicates/proofs.py

```
class Proof:
        ⋮
    class MPLine:
            ⋮
        def is_valid(self, assumptions: AbstractSet[Schema],
                     lines: Sequence[Proof.Line], line_number: int) -> bool:
            """Checks if the current line is validly justified in the context of
            the specified proof.

            Parameters:
                assumptions: assumptions/axioms of the proof.
                lines: lines of the proof.
                line_number: line number of the current line in the given lines.

            Returns:
                ``True`` if the formula of the line from the given lines whose
                number is the conditional line number justifying the current
                line is '(`antecedent`->`consequent`)', where `antecedent` is
                the formula of the line from the given lines whose number is the
                antecedent line number justifying the current line and
                `consequent` is the formula justified by the current line;
                ``False`` otherwise.
            """
            assert line_number < len(lines) and lines[line_number] is self
            # Task 9.6
```

Hint: Recall again that this method takes the three arguments `assumptions`, `lines`, and `line_number` in order to take the same arguments as the `is_valid()` methods of other proof line classes. You need not use all of these in your solution to this task.

2.3 Universal Generalization (UG) Lines

Finally, the class `Proof.UGLine` is used for proof lines justified by the UG inference rule. This class holds, in addition to the formula that it justifies, also the line number of the previous line from which this formula is deduced via UG.

predicates/proofs.py

```
class Proof:
        ⋮
    @frozen
    class UGLine:
        """An immutable proof line justified by the Universal Generalization
        (UG) inference rule.

        Attributes:
            formula: the formula justified by the line.
            predicate_line_number: the line number of the predicate of the UG
                inference justifying the line.
        """
        formula: Formula
        predicate_line_number: int

        def __init__(self, formula: Formula, predicate_line_number: int):
            """Initializes a `Proof.UGLine` from its formula and line number of
            the predicate of the UG inference justifying it.

            Parameters:
                formula: the formula to be justified by the line.
                predicate_line_number: the line number of the predicate of the
                    UG inference justifying the line.
            """
            self.formula = formula
            self.predicate_line_number = predicate_line_number
```

Task 7. Implement the missing code for the method `is_valid(assumptions, lines, line_number)` of class `Proof.UGLine`, which returns whether the formula of the current line is validly justified via an application of UG to the specified previous line.

predicates/proofs.py

```
class Proof:
        ⋮
    class UGLine:
            ⋮
        def is_valid(self, assumptions: AbstractSet[Schema],
                     lines: Sequence[Proof.Line], line_number: int) -> bool:
            """Checks if the current line is validly justified in the context of
            the specified proof.

            Parameters:
                assumptions: assumptions/axioms of the proof.
                lines: lines of the proof.
                line_number: line number of the current line in the given lines.

            Returns:
                ``True`` if the formula of the current line is of the form
                'A`x`[`predicate`]', where `predicate` is the formula of the
                line from the given lines whose number is the predicate line
```

DRAFT

```
                number justifying the current line and `x` is any variable name;
                ``False`` otherwise.
            """
            assert line_number < len(lines) and lines[line_number] is self
            # Task 9.7
```

Hint: Recall once more that this method takes the three arguments `assumptions`, `lines`, and `line_number` in order to take the same arguments as the `is_valid()` methods of other proof line classes. You need not use all of these in your solution to this task.

2.4 Tautology Lines

We now finally give a precise definition for what a predicate-logic tautology is. Predicate-logic formulas generalize propositional formulas by replacing the propositional atoms with structured subformulas whose root is a relation, an equality, or a quantifier. We define the **propositional skeleton** of a predicate-logic formula as the propositional formula that consistently replaces each of these subformulas 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 call a predicate-logic formula a **(predicate-logic) tautology** if its propositional skeleton (a propositional formula) is a tautology.[7] The class `Proof.TautologyLine` is used for proof lines justified as predicate-logic tautologies.

predicates/proofs.py

```
class Proof:
        ⋮
    @frozen
    class TautologyLine:
        """An immutable proof line justified as a tautology.

        Attributes:
            formula: the formula justified by the line.
        """
        formula: Formula

        def __init__(self, formula: Formula):
            """Initializes a `Proof.TautologyLine` from its formula.

            Parameters:
                formula: the formula to be justified by the line.
            """
            self.formula = formula
```

Task 8. Implement the missing code for the method `propositional_skeleton()` of class `Formula` (in the file `predicates/syntax.py`), which returns a propositional skeleton of the current (predicate-logic) formula—an object of type

[7] We use the terminology *the* propositional skeleton somewhat misleadingly, as there are many propositional skeletons for any given predicate-logic formula. For example, '((z3|z4)→z3)' is also a propositional skeleton of the formula '(R(x)|Q(y))→R(x))'. There is no problem here, though, since either all skeletons of a given formula are tautologies, or none are.

propositions.syntax.Formula[8]—along with the map from propositional variable names of the returned skeleton (e.g., 'z8') to the predicate-logic subformulas of the current formula that they have replaced.

predicates/syntax.py

```
class Formula:
        ⋮
    def propositional_skeleton(self) -> Tuple[PropositionalFormula,
                                              Mapping[str, Formula]]:
        """Computes a propositional skeleton of the current formula.

        Returns:
            A pair. The first element of the pair is a propositional formula
            obtained from the current formula by substituting every (outermost)
            subformula that has a relation or quantifier at its root with an
            atomic propositional formula, consistently such that multiple equal
            such (outermost) subformulas are substituted with the same atomic
            propositional formula. The atomic propositional formulas used for
            substitution are obtained, from left to right, by calling
            `next(fresh_variable_name_generator)`. The second element of the
            pair is a mapping from each atomic propositional formula to the
            subformula for which it was substituted.

        Examples:
            >>> formula = Formula.parse('((Ax[x=7]&x=7)|(x=7->~Q(y)))')
            >>> formula.propositional_skeleton()
            (((z1&z2)|(z2->~z3)), {'z1': Ax[x=7], 'z2': x=7, 'z3': Q(y)})
            >>> formula.propositional_skeleton()
            (((z4&z5)|(z5->~z6)), {'z4': Ax[x=7], 'z5': x=7, 'z6': Q(y)})
        """
        # Task 9.8
```

Guidelines: The atomic propositional formulas in this propositional formula should be named 'z1', 'z2', ..., ordered according to their first (left-most) occurrence in the original predicate-logic 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 9. Implement the missing code for the method is_valid(assumptions, lines, line_number) of class Proof.TautologyLine, which returns whether the formula of the current line really is a predicate-logic tautology.

predicates/proofs.py

```
class Proof:
        ⋮
    class TautologyLine:
                ⋮
        def is_valid(self, assumptions: AbstractSet[Schema],
                     lines: Sequence[Proof.Line], line_number: int) -> bool:
            """Checks if the current line is validly justified in the context of
            the specified proof.

            Parameters:
```

[8]To avoid naming conflicts, this class is imported for you into predicates/syntax.py under the name PropositionalFormula.

```
            assumptions: assumptions/axioms of the proof.
            lines: lines of the proof.
            line_number: line number of the current line in the given lines.

        Returns:
            ``True`` if the formula justified by the current line is a
            (predicate-logic) tautology, ``False`` otherwise.
        """
        assert line_number < len(lines) and lines[line_number] is self
        # Task 9.9
```

Hint: Use Task 8. The function `propositions.semantics.is_tautology()` is imported for you into `predicates/proofs.py` under the name `is_propositional_tautology()`. Recall yet again that this method takes the three arguments `assumptions`, `lines`, and `line_number` in order to take the same arguments as the `is_valid()` methods of other proof line classes. You need not use these in your solution to this task.

As already noted, we allow for the use of predicate-logic tautologies without proof purely for simplicity. Indeed, by proving all needed tautologies and "inlining" the proofs we could have done away with tautology justifications, as well as with their *semantic* validation, resulting in purely syntactic validation of proofs, as one may desire. We will discuss (an implement) this in Section 3 below.

2.5 The Soundness of Proofs

Similarly to Propositional Logic, we will say that a formula ϕ is **provable** from a set of schemas A, and write $A \vdash \phi$, if there is a (valid) proof of ϕ from the assumptions/axioms A (using assumption lines, MP lines, UG lines, and tautology lines). Very similarly to Propositional Logic, it is not hard to prove that the above proof system is sound.

Definition (Sound Inference)**.** We say that a set of assumptions A **entails** a conclusion ϕ if every model that satisfies each of the assumptions in A (for any assignment to its free variables) also satisfies ϕ (for any assignment to its free variables). We denote this by $A \models \phi$.[9] We say that the inference of ϕ from A is **sound** if $A \models \phi$. If A is the empty set then we simply write $\models \phi$, which is equivalent to saying that ϕ is sound (as defined on page 147).

Theorem (The Soundness Theorem for Predicate Logic)**.** *Any inference that is provable via (only) sound assumptions/axioms is sound. That is, if X contains only sound schemas, and if $A \cup X \vdash \phi$, then $A \models \phi$.*

As programmatically proving the Soundness Theorem for Predicate Logic would not add any new insights beyond your corresponding programmatic proof of the Soundness Theorem for Propositional Logic, we will skip the involved programming, and instead prove the soundness of the Predicate Logic in the traditional mathematical way, using induction over the lines of a proof.

[9]As we have remarked also in the first part of this book, the symbol $\models$ is sometimes used also in a slightly different way: for a model M and a formula ϕ one may write $M \models \phi$ (i.e., M is a model of ϕ) to mean that ϕ evaluates to *True* in the model M.

Proof. Fix a model for A. All that we need to verify is that each type of justification that we allow results in a conclusion that holds in the model if the formula of each previous line holds in the model:

- Assumption line: Any schema in A holds in the model by definition of the model. Any schema in X is sound, and so by definition any of their instances is sound and so holds in any model.

- MP line: The reasoning is similar yet a bit more detailed. Fix an assignment to the free variables of the conclusion of the line. Arbitrarily augment the assignment with meanings for any additional free variables that occur in the two justifying formulas (from previous lines)—this does not change the truth value of the conclusion in the model with that assignment. By definition, each of the justifying formulas holds in the model with the augmented assignment, so by the semantic definition of the evaluation of the implication operator, so does the conclusion. So, the conclusion holds in the model with the original assignment.

- UG line: The justifying formula (from a previous line) holds in the model for any assignment to its free variables, and in particular for any assignment to free occurrences of the variable over which UG is taken, and that is precisely also the semantic meaning of the quantification in the conclusion of the UG line.[10]

- Tautology line: Any assignment to the free variables of the conclusion of the line defines, together with the model, an assignment of a truth value to each subformula whose root is a relation, equality, or quantifier. This corresponds to an assignment of a truth value to each of the propositional atoms of the skeleton of the conclusion. Since the skeleton evaluates to *True* under the latter assignment (as it is a propositional tautology), so does the former in the model with the original assignment. □

We conclude this section with a brief discussion of why we have chosen (as is customary) to treat free variables as universally quantified in formulas. First, why allow free variables at all? Well, to avoid clutter. Indeed, we could have done just as well without allowing free variables in formulas if we would have, e.g., allowed every universal closure of a tautology, that is, every formula of the form '$\forall p_1[\forall p_1[\cdots\forall p_n[\phi]\cdots]]$' where ϕ is a tautology and the quantifications are over all of the free variables of ϕ, but then, to avoid losing expressive power, we would have had to define MP in a much more bulky way: to allow not only to deduce, e.g., `~(R(x,y)→Q(x,y))` from `R(x,y)` and from `~Q(x,y)`, but also to deduce, e.g., `∀x[∀y[~(R(x,y)→Q(x,y))]` from `∀x[∀y[R(x,y)]` and from `∀x[∀y[~Q(x,y)]`. Given this example, it is easy to see that choosing to allow free variables and to treat them as universally quantified, in combination with allowing UG, gives a far simpler definition of proof justifications. Second, why do we treat free variables as universally quantified and not, say, existentially quantified? Well, since in that case, while tautologies would still be valid (well, at least as long as there is at least one element in the universe), the above usage of MP would for example not be valid. Indeed, deducing `∃x[∃y[~(R(x,y)→Q(x,y))]` from `∃x[∃y[R(x,y)]` and from `∃x[∃y[~Q(x,y)]]` is fundamentally flawed (make sure that you understand why). Very similarly, in the next section we will see that treating free variables as universally quantified allows us to easily translate a propositional-logic proof of the (propositional tautology) skeleton

[10] This argument clarifies the meaning of our cryptic comment above that UG syntactically encompasses our *semantic* treatment of free variables as being universally quantified.

DRAFT

of a predicate-logic tautology into a predicate-logic proof for that predicate-logic tautology, which in fact allows us to translate any predicate-logic proof into a proof without tautology line justifications.

3 Getting Rid of Tautology Lines

We conclude this chapter by showing that the ability to use tautologies as building blocks in our proof does not really give our proofs more proving power (and so is for convenience). That is, in this section you will show that any tautology is provable using only assumption and MP line justifications via a set of schemas that correspond to our axiomatic system for Propositional Logic. For convenience, we will focus in this section only on tautologies that contain only the implication and negation operators, and will therefore only require schemas that correspond to our axiomatic system for implication and negation. These schemas are defined in `predicates/proofs.py`.

predicates/proofs.py

```
# Schema equivalents of the propositional-logic axioms for implication and
# negation

#: Schema equivalent of the propositional-logic self implication axiom `I0`.
I0_SCHEMA = Schema(Formula.parse('(P()->P())'), {'P'})
#: Schema equivalent of the propositional-logic implication introduction (right)
#: axiom `I1`.
I1_SCHEMA = Schema(Formula.parse('(Q()->(P()->Q()))'), {'P', 'Q'})
#: Schema equivalent of the propositional-logic self-distribution of implication
#: axiom `D`.
D_SCHEMA = Schema(Formula.parse(
    '((P()->(Q()->R()))->((P()->Q())->(P()->R())))'), {'P', 'Q', 'R'})
#: Schema equivalent of the propositional-logic implication introduction (left)
#: axiom `I2`.
I2_SCHEMA = Schema(Formula.parse('(~P()->(P()->Q()))'), {'P', 'Q'})
#: Schema equivalent of the propositional-logic converse contraposition axiom
#: `N`.
N_SCHEMA  = Schema(Formula.parse('((~Q()->~P())->(P()->Q()))'), {'P', 'Q'})
#: Schema equivalent of the propositional-logic negative-implication
#: introduction axiom `NI`.
NI_SCHEMA = Schema(Formula.parse('(P()->(~Q()->~(P()->Q())))'), {'P', 'Q'})
#: Schema equivalent of the propositional-logic double-negation introduction
#: axiom `NN`.
NN_SCHEMA = Schema(Formula.parse('(P()->~~P())'), {'P'})
#: Schema equivalent of the propositional-logic resolution axiom `R`.
R_SCHEMA  = Schema(Formula.parse(
    '((Q()->P())->((~Q()->P())->P()))'), {'P', 'Q'})

#: Schema system equivalent of the axioms of the propositional-logic large
#: axiomatic system for implication and negation `AXIOMATIC_SYSTEM`.
PROPOSITIONAL_AXIOMATIC_SYSTEM_SCHEMAS = {I0_SCHEMA, I1_SCHEMA, D_SCHEMA,
                                          I2_SCHEMA, N_SCHEMA, NI_SCHEMA,
                                          NN_SCHEMA, R_SCHEMA}

#: Mapping from propositional-logic axioms for implication and negation to their
#: schema equivalents.
PROPOSITIONAL_AXIOM_TO_SCHEMA = {
        I0: I0_SCHEMA, I1: I1_SCHEMA, D: D_SCHEMA, I2: I2_SCHEMA, N: N_SCHEMA,
        NI: NI_SCHEMA, NN: NN_SCHEMA, R: R_SCHEMA}
```

Lemma. *All of the above schemas are sound.*

Proof. It is straightforward to see that each instance of these schemas is a tautology, so the same reasoning as in the proof of the Soundness Theorem above applies. □

Our first step toward showing how to prove any tautology (without tautology line justifications of course) from the above schemas is to transform a propositional skeleton of a predicate-logic formula into that predicate-logic formula using a given substitution map.

Task 10. Implement the missing code for the static method `from_propositional_skeleton(skeleton, substitution_map)` of class `Formula`, which returns a predicate-logic formula `formula` such that the pair `(skeleton, substitution_map)` is a legal return value of `formula.propositional_skeleton()`.

predicates/syntax.py

```
class Formula:
        ⋮
    @staticmethod
    def from_propositional_skeleton(skeleton: PropositionalFormula,
                                    substitution_map: Mapping[str, Formula]) -> \
            Formula:
        """Computes a predicate-logic formula from a propositional skeleton and
        a substitution map.

        Arguments:
            skeleton: propositional skeleton for the formula to compute,
                containing no constants or operators beyond '~', '->', '|', and
                '&'.
            substitution_map: mapping from each atomic propositional subformula
                of the given skeleton to a predicate-logic formula.

        Returns:
            A predicate-logic formula obtained from the given propositional
            skeleton by substituting each atomic propositional subformula with
            the formula mapped to it by the given map.

        Examples:
            >>> Formula.from_propositional_skeleton(
            ...     PropositionalFormula.parse('((z1&z2)|(z2->~z3))'),
            ...     {'z1': Formula.parse('Ax[x=7]'), 'z2': Formula.parse('x=7'),
            ...      'z3': Formula.parse('Q(y)')})
            ((Ax[x=7]&x=7)|(x=7->~Q(y)))
        """
        for operator in skeleton.operators():
            assert is_unary(operator) or is_binary(operator)
        for variable in skeleton.variables():
            assert variable in substitution_map
        # Task 9.10
```

Our next and main step is to transform a proof of a propositional skeleton of a predicate-logic formula into a proof for that formula.

Task 11.

1. Implement the missing code for the function `_axiom_specialization_map_to_schema_instantiation_map(propositional_specialization_map, substitution_map)`, which takes a specialization map that specifies how a propositional-logic axiom `a` from `AXIOMATIC_SYSTEM` specializes into some specialization `s`, and a substitution map from the propositional atoms of `s` to predicate-logic formulas that was returned by some call to `propositional_skeleton`, and returns an instantiation map for instantiating the schema equivalent of `a` (e.g., if `a` is I0, then its schema equivalent is I0_SCHEMA) into a predicate-logic formula `formula` whose skeleton is `s`, such that the pair `(s,substitution_map)` is a legal return value of a call to `formula.propositional_skeleton()`.

predicates/proofs.py

```
def _axiom_specialization_map_to_schema_instantiation_map(
        propositional_specialization_map: PropositionalSpecializationMap,
        substitution_map: Mapping[str, Formula]) -> Mapping[str, Formula]:
    """Converts the given propositional-logic specialization map from a
    propositional-logic axiom to its specialization, to an instantiation map
    from the schema equivalent of that axiom to a predicate-logic formula whose
    skeleton is that specialization.

    Parameters:
        propositional_specialization_map: mapping specifying how some
            propositional-logic axiom `axiom` (which is not specified) from
            `AXIOMATIC_SYSTEM` specializes into some specialization
            `specialization` (which is also not specified), and containing no
            constants or operators beyond '~', '->', '|', and '&'.
        substitution_map: mapping from each atomic propositional subformula of
            `specialization` to a predicate-logic formula.

    Returns:
        An instantiation map for instantiating the schema equivalent of `axiom`
        into the predicate-logic formula obtained from its propositional
        skeleton `specialization` by the given substitution map.

    Examples:
        >>> _axiom_specialization_map_to_schema_instantiation_map(
        ...     {'p': PropositionalFormula.parse('(z1->z2)'),
        ...      'q': PropositionalFormula.parse('~z1')},
        ...     {'z1': Formula.parse('Ax[(x=5&M())]'),
        ...      'z2': Formula.parse('R(f(8,9))')})
        {'P': (Ax[(x=5&M())]->R(f(8,9))), 'Q': ~Ax[(x=5&M())]}
    """
    for variable in propositional_specialization_map:
        assert is_propositional_variable(variable)
        for operator in propositional_specialization_map[variable].operators():
            assert is_unary(operator) or is_binary(operator)
    for key in substitution_map:
        assert is_propositional_variable(key)
    # Task 9.11.1
```

Hint: You can assume that the keys of `propositional_specialization_map` are a subset of `{'p', 'q', 'r'}`.

2. Implement the missing code for the function `_prove_from_skeleton_proof(formula, skeleton_proof, substitution_map)`, which takes a predicate-logic

DRAFT

formula `formula`, a propositional-logic proof[11] of a propositional skeleton `skeleton` of `formula` via `AXIOMATIC_SYSTEM`, and a substitution map such that the pair `(skeleton,substitution_map)` is a legal return value of a call to `formula.propositional_skeleton()`. This function returns a predicate-logic proof of `formula` via `PROPOSITIONAL_AXIOMATIC_SYSTEM_SCHEMAS` that contains only assumption and MP lines.

predicates/proofs.py

```
def _prove_from_skeleton_proof(formula: Formula,
                               skeleton_proof: PropositionalProof,
                               substitution_map: Mapping[str, Formula]) -> \
        Proof:
    """Converts the given proof of a propositional skeleton of the given
    predicate-logic formula to a proof of that predicate-logic formula.

    Parameters:
        formula: predicate-logic formula to prove.
        skeleton_proof: valid propositional-logic proof of a propositional
            skeleton of the given formula, from no assumptions and via
            `AXIOMATIC_SYSTEM`, and containing no constants or operators beyond
            '~', '->', '|', and '&'.
        substitution_map: mapping from each atomic propositional subformula of
            the skeleton of the given formula that is proven in the given proof
            to the respective predicate-logic subformula of the given formula.

    Returns:
        A valid predicate-logic proof of the given formula from the axioms
        `PROPOSITIONAL_AXIOMATIC_SYSTEM_SCHEMAS` via only assumption lines and
        MP lines.
    """
    assert len(skeleton_proof.statement.assumptions) == 0 and \
           skeleton_proof.rules.issubset(PROPOSITIONAL_AXIOMATIC_SYSTEM) and \
           skeleton_proof.is_valid()
    assert Formula.from_propositional_skeleton(
        skeleton_proof.statement.conclusion, substitution_map) == formula
    for line in skeleton_proof.lines:
        for operator in line.formula.operators():
            assert is_unary(operator) or is_binary(operator)
    # Task 9.11.2
```

Guidelines: Since there are no assumptions in the given proof, each line is either the result of an application of MP to previous lines, or a specialization of an axiom. In either case, use your solution to Task 10; in the latter case use also your solution to the first part of this task.

Hint: To allow you to use the method `formula_specialization_map()` of class `propositions.proofs.InferenceRule` while maintaining readability, that class is imported for you into `predicates/proofs.py` under the name `PropositionalInferenceRule`.

You are now in good shape to use your solution of Task 11 to show that every predicate-logic tautology is provable via the above schemas.

[11] To avoid naming conflicts, the class `propositions.proofs.Proof` is imported for you into `predicates/proofs.py` under the name `PropositionalProof`.

DRAFT

Task 12 (Programmatic Proof of the Predicate Logic Version of the Tautology Theorem)**.** Implement the missing code for the function `prove_tautology(tautology)`, which proves the given predicate-logic tautology from the axioms `PROPOSITIONAL_AXIOMATIC_SYSTEM_SCHEMAS` with only assumption and MP lines.

predicates/proofs.py
```
def prove_tautology(tautology: Formula) -> Proof:
    """Proves the given predicate-logic tautology.

    Parameters:
        tautology: predicate-logic tautology to prove.

    Returns:
        A valid proof of the given predicate-logic tautology from the axioms
        `PROPOSITIONAL_AXIOMATIC_SYSTEM_SCHEMAS` via only assumption lines
        and MP lines.
    """
    assert is_propositional_tautology(tautology.propositional_skeleton()[0])
    # Task 9.12
```

Hint: To avoid naming conflicts, the function `propositions.tautology.prove_tautology()` is imported for you into `predicates/proofs.py` under the name `prove_propositional_tautology()`.

Your solution to Task 12 programmatically proves the following theorem.

Theorem (The Tautology Theorem: Predicate Logic Version)**.** *For every predicate-logic tautology ϕ, there exists a valid proof of ϕ that uses only assumption and MP line justifications (and not tautology or UG line justifications), from the schema equivalents of $\mathcal{H}$ (or from the schema equivalents of $\widehat{\mathcal{H}}$ if the tautology involves Boolean operators beyond implication and negation).*

We therefore obtain that indeed tautology line justifications can be dropped from our proofs without losing any proving power (as long as we add a few schemas).

Corollary. *If a predicate-logic formula ϕ is provable from a set of assumptions/axioms A, then it is provable without tautology line justifications from A as well as the schema equivalents of $\widehat{\mathcal{H}}$.*

DRAFT