

Chapter 12:

The Completeness Theorem

In this chapter, in which the analysis of this entire book culminates, we will take the final steps toward proving the **Completeness Theorem** for Predicate Logic—the main theorem of Predicate Logic, which states that a set of formulas is **consistent** if and only if it has a model. Similarly to Propositional Logic, we call a set of formulas in Predicate Logic **consistent** if a contradiction (i.e., the negation of a tautology, such as $(R(x) \wedge \neg R(x))$) cannot be proven from it and from our six axiom schemas (equivalently, from it and from these six schemas plus the sixteen additional schemas from Chapter 11).¹ So, consistency is a syntactic concept, while having a model is a semantic one, and the Completeness Theorem relates these two seemingly very different concepts. While we will not be able to create a fully programmatic proof for the Completeness Theorem (we will have to complete the very last step mathematically, as it involves infinite models), we will be able to programmatically prove almost all of the required lemmas, and to describe the general flow of the mathematical proof.

As in Propositional Logic, there is an “easy direction” to the Completeness Theorem that follows from (or rather, is a restatement of) the Soundness Theorem for Predicate Logic. Recall that the Soundness Theorem for Predicate Logic from Chapter 9 states that if a set of formulas A can be used, alongside our (basic and additional) twenty-two (sound) axiom schemas, to prove a formula ϕ , then ϕ holds in any model where all of A hold. In particular, if A has a model, then ϕ must hold in that model, so ϕ cannot be a contradiction, and so A is consistent. This chapter is therefore dedicated to proving the “hard direction” of the Completeness Theorem: that every consistent set of formulas has a model.

In this chapter, we will assume that we are in a setting without any function names and without the equality symbol—as you have shown in Chapter 8, this does not make Predicate Logic lose any expressive power. Furthermore, it will suffice to carry out the core of our proof with formulas that are given in prenex normal form, since you have shown in Chapter 11 that this does not lose any generality. Finally, it will also be convenient to work with formulas that are actually **sentences**, i.e., have no free variable names. Again, this is without loss of generality since as we have seen, we can go back and forth between sentences and formulas by universally quantifying over any free variable name using UG on the one hand, and by removing universal quantifications using UI (replacing any quantified variable name “with itself”) on the other hand. So, we wish to show that every consistent

¹Recall from Propositional Logic that we call a set inconsistent if the negation of an axiom is provable from it via our axiomatic system. In Propositional Logic we defined this with respect to the axiom I0 (but saw that defining this with respect to any other axiom would have been equivalent), and in Predicate Logic we define this with respect to any tautology, since we allow all tautologies as axioms in our proof thanks to your proof of the Tautology Theorem in Propositional Logic (and once again, defining this with respect to any other axiom would have been an equivalent completely syntactic definition). This is completely analogous to our reinterpretation of proofs by way of contradiction for Predicate Logic in Chapter 11 as compared to Chapter 5.

set of sentences in prenex normal form (that, as will always be the case in this chapter, are without function names or equalities) has a model. All of the functions that you are asked to implement in this chapter are contained in the file `predicates/completeness.py`.

Since we have “assumed away” function names, to construct a model of a given consistent set of sentences we will need to figure out three things:

1. which universe to use for the model,
2. which interpretation to give for each constant name, i.e., how to map each constant name to an element from the universe, and
3. which interpretation to give for each relation name, i.e., for every relation name R and every tuple of elements from the universe $(\alpha_1, \dots, \alpha_n)$ where n is the arity of R , whether the tuple $(\alpha_1, \dots, \alpha_n)$ is in the interpretation of R .

Since we have “assumed away” the equality symbol, we will handle the first two of these questions (which universe to use, and which interpretation to give for each constant name) in a very simple (yet quite ingenious) manner: our universe will be the set of the constant names that we use, and the interpretation of each constant name (when viewed as a constant name) will be the constant name itself (when viewed as an element of the universe).²

To handle the third of these questions (regarding which interpretation to give for each relation name), it would be quite convenient if for each relation name R , for every tuple of constant names (c_1, \dots, c_n) where n is the arity of R , the given consistent set of sentences already had in it either the **primitive** sentence ‘ $R(c_1, \dots, c_n)$ ’ or its negation ‘ $\sim R(c_1, \dots, c_n)$ ’ (the set of sentences cannot contain both since then it would not be consistent), since then we could simply “read off” the interpretation of R from such sentences: if the set of sentences had this primitive sentence (without any negation), then the tuple (c_1, \dots, c_n) should be in the interpretation of R , while if the set of sentences had the negation of this primitive sentence, then this tuple should *not* be in the interpretation of R . The basic strategy of our proof of the (“hard direction” of the) Completeness Theorem will therefore be to add enough sentences (and in particular, enough sentences that are either primitive sentences or the negation of primitive sentences) to the given set of sentences—*while leaving it consistent*—so that we will indeed be able to “read off” the interpretations of all relation names from such primitive sentences, or from their negations, in the set. In particular, we will consider ourselves to have added “enough” sentences to the given set if after having added them, the set satisfies the following condition:

Definition (Closed Set of Sentences; Primitively Closed Set of Sentences; Universally Closed Set of Sentences; Existentially Closed Set of Sentences). Let S be a (possibly infinite) set of sentences in prenex normal form. We say that S is **closed** if all of the following hold:

- S is **primitively closed**: for every relation name R that appears somewhere in S and for every tuple of (not necessarily distinct) constant names (c_1, \dots, c_n) where n is the arity of R and each c_i appears somewhere in S , either the **primitive** sentence ‘ $R(c_1, \dots, c_n)$ ’ or its negation ‘ $\sim R(c_1, \dots, c_n)$ ’ (or both) is in S .

²Had we not assumed away the equality symbol, we would have had to be far more careful here and take the elements of the universe to be equivalence classes of constant names, similarly to our construction from Chapter 8. The approach that we have adopted therefore allows for that argument to be handled in a separate proof (that we have already given in Chapter 8), instead of being intertwined into our proof of the Completeness Theorem.

- S is **universally closed**: for every universally quantified sentence $\forall x[\phi(x)]$ in S and every constant name c that appears somewhere in S , the **universal instantiation** $\phi(c)$ is also in S .³
- S is **existentially closed**: for every existentially quantified sentence $\exists x[\phi(x)]$ in S , there exists a constant name c such that the **existential witness**⁴ $\phi(c)$ is also in S .

The function `is_closed(sentences)` takes a (finite) set of sentences in prenex normal form and returns whether this set is closed. While we have already implemented this function for you, it is missing a few core components, which you will implement in the next task.

predicates/completeness.py

```
def is_closed(sentences: AbstractSet[Formula]) -> bool:
    """Checks whether the given set of prenex-normal-form sentences is closed.

    Parameters:
        sentences: set of prenex-normal-form sentences to check.

    Returns:
        ``True`` if the given set of sentences is primitively, universally, and
        existentially closed; ``False`` otherwise.
    """
    for sentence in sentences:
        assert is_in_prenex_normal_form(sentence) and \
            len(sentence.free_variables()) == 0
    return is_primitively_closed(sentences) and \
        is_universally_closed(sentences) and \
        is_existentially_closed(sentences)
```

The function `get_constants(formulas)`, which takes a (finite) set of formulas and returns the set of constant names that appear in these formulas, will be useful in implementing the three missing core components of the function `is_closed()` in the next task. We have already implemented this function for you using the method `constants()` of class `Formula` that you have implemented in Chapter 7.

predicates/syntax.py

```
def get_constants(formulas: AbstractSet[Formula]) -> Set[str]:
    """Finds all constant names in the given formulas.

    Parameters:
        formulas: formulas to find all constant names in.

    Returns:
        A set of all constant names used in one or more of the given formulas.
    """
    constants = set()
    for formula in formulas:
```

³Here and below, similarly to the notation used in previous chapters, by $\phi(x)$ we mean any formula that may have x as a free variable name (but may have no other free variable names, as otherwise $\forall x[\phi(x)]$ would not be a sentence), and by $\phi(c)$ we mean the sentence obtained from $\phi(x)$ by replacing every free occurrence of x with c .

⁴The sentence $\phi(c)$ is called an **existential witness** to $\exists x[\phi(x)]$ since if the former holds in a model, then it “witnesses” that there really *exists* an element of the universe (the interpretation of c) such that if it is assigned to x then $\phi(x)$ holds in that model.

```

    constants.update(formula.constants())
    return constants

```

Task 1. Implement the missing components of the function `is_closed()`, that is:

- a. Implement the missing code for the function `is_primitively_closed(sentences)`, which returns whether the given (finite) set of sentences in prenex normal form is primitively closed.

predicates/completeness.py

```

def is_primitively_closed(sentences: AbstractSet[Formula]) -> bool:
    """Checks whether the given set of prenex-normal-form sentences is
    primitively closed.

    Parameters:
        sentences: set of prenex-normal-form sentences to check.

    Returns:
        ``True`` if for every n-ary relation name from the given sentences, and
        for every n (not necessarily distinct) constant names from the given
        sentences, either the invocation of this relation name over these
        constant names (in order), or the negation of this invocation (or both),
        is one of the given sentences; ``False`` otherwise.
    """
    for sentence in sentences:
        assert is_in_prenex_normal_form(sentence) and \
            len(sentence.free_variables()) == 0
    # Task 12.1a

```

Hint: The `relations()` method of class `Formula` may be useful here, and so may the `product()` method (with its `repeat` argument) from the standard Python `itertools` module.

- b. Implement the missing code for the function `is_universally_closed(sentences)`, which returns whether the given (finite) set of sentences in prenex normal form is universally closed.

predicates/completeness.py

```

def is_universally_closed(sentences: AbstractSet[Formula]) -> bool:
    """Checks whether the given set of prenex-normal-form sentences is
    universally closed.

    Parameters:
        sentences: set of prenex-normal-form sentences to check.

    Returns:
        ``True`` if for every universally quantified sentence from the given set
        of sentences, and for every constant name from these sentences, the
        statement quantified by this sentence, with every free occurrence of the
        universal quantification variable name replaced with this constant name,
        is also in the given set; ``False`` otherwise.
    """
    for sentence in sentences:
        assert is_in_prenex_normal_form(sentence) and \
            len(sentence.free_variables()) == 0
    # Task 12.1b

```

Hint: The `substitute()` method of class `Formula` may be useful here.

- c. Implement the missing code for the function `is_existentially_closed(sentences)`, which returns whether the given (finite) set of sentences in prenex normal form is existentially closed.

predicates/completeness.py

```
def is_existentially_closed(sentences: AbstractSet[Formula]) -> bool:
    """Checks whether the given set of prenex-normal-form sentences is
    existentially closed.

    Parameters:
        sentences: set of prenex-normal-form sentences to check.

    Returns:
        ``True`` if for every existentially quantified sentence from the given
        set of sentences there exists a constant name such that the statement
        quantified by this sentence, with every free occurrence of the
        existential quantification variable name replaced with this constant
        name, is also in the given set; ``False`` otherwise.
    """
    for sentence in sentences:
        assert is_in_prenex_normal_form(sentence) and \
            len(sentence.free_variables()) == 0
    # Task 12.1c
```

Hint: The `substitute()` method of class `Formula` may be useful here as well.

Our strategy for the remainder of this chapter will be as follows. In Section 1, we will show that once we have a closed set of sentences S then either we will be able to create a model of it that has the set of constant names that appear in S as the universe of the model (by “reading off” the interpretations of all relation names from S , as outlined above), or otherwise we will be able to explicitly prove a contradiction from S (showing S to be inconsistent). In Section 2, we will show that any set of sentences S can have sentences added to it in a way that makes it closed, yet leaves it consistent if it originally were consistent. In Sections 3 and 4 we will state and discuss the Completeness Theorem and alternate versions thereof, as well as other conclusions and consequences of this important theorem.

1 Deriving a Model or a Contradiction for a Closed Set

In this section, we will show that once we have a closed set of sentences S , then either we will be able to create a model of it with the set of constant names that appear in S as its universe, or, if we fail in creating such a model, then we will show how to explicitly prove a contradiction from S (showing that it is in fact inconsistent). That is, in this section we will prove the following lemma:

Lemma (Completeness for Closed Sets). *Let S be a (possibly infinite) closed set of sentences in prenex normal form. If S is consistent, then S has a model.*

We will start by attempting to construct a model of S with the set of constant names that appear in S as its universe. Specifically, if the primitive sentence ' $R(c_1, \dots, c_n)$ ' is contained in S then any such model of S must have the tuple (c_1, \dots, c_n) in the interpretation of R , and so we will add this tuple to this interpretation in the model that we are creating. Conversely, if ' $\sim R(c_1, \dots, c_n)$ ' is in S , then this tuple must *not* be in the interpretation of R in any such model of S , so we will *not* add this tuple to this interpretation in the model that we are creating. (If both of these sentences are in S , then S is inconsistent since the contradiction ' $(R(c_1, \dots, c_n) \& \sim R(c_1, \dots, c_n))$ ' is provable from it, and we are done.) Due to the primitive closure condition, S is assured to contain one of these primitive sentences for every relation name R and every tuple of constant names that each appear somewhere in S , so the model that we are creating is uniquely determined, and is in fact the only possible candidate (among those whose universe is the set of constant names that appear somewhere in S) for being a model of S . If this model satisfies all of the sentences in S , then we are done. Otherwise, some sentence in S is unsatisfied by the model, and we will want to show that S is in fact inconsistent. Our first order of business is to show that due to the universal and existential closure conditions, one can remove all of the quantifiers from this unsatisfied sentence, to find a quantifier-free sentence that is unsatisfied by the model.

Task 2. Implement the missing code for the function `find_unsatisfied_quantifier_free_sentence(sentences, model, unsatisfied)`, which takes a universally and existentially closed (finite) “set” (see below) of sentences in prenex normal form `sentences`, takes a model `model` whose universe is the set of constant names that appear somewhere in `sentences`, and takes a sentence (which possibly contains quantifiers) from `sentences` that `model` does not satisfy, and returns a quantifier-free sentence from `sentences` that `model` does not satisfy. To verify that you indeed use the universal and existential closure conditions in your solution (and not simply iterate over `sentences` until you find a suitable sentence to return), the tests for this task are implemented such that the “set” `sentences` that this function is given in fact may not be iterated over. Instead, it may only be accessed using containment queries, i.e., using the Python `in` operator as in: `if sentence in sentences`. (In Python terminology, the parameter `sentences` is a `Container` but not an `AbstractSet`.)

predicates/completeness.py

```
def find_unsatisfied_quantifier_free_sentence(sentences: Container[Formula],
                                             model: Model[str],
                                             unsatisfied: Formula) -> Formula:
    """
    Given a universally and existentially closed set of prenex-normal-form
    sentences, given a model whose universe is the set of all constant names
    from the given sentences, and given a sentence from the given set that the
    given model does not satisfy, finds a quantifier-free sentence from the
    given set that the given model does not satisfy.

    Parameters:
        sentences: universally and existentially closed set of
                   prenex-normal-form sentences, which is only to be accessed using
                   containment queries, i.e., using the ``in`` operator as in:

    >>> if sentence in sentences:
    ...     print('contained!')
```

```

    model: model for all element names from the given sentences, whose
           universe is `get_constants(sentences)`.
    unsatisfied: sentence (which possibly contains quantifiers) from the
                  given sentences that is not satisfied by the given model.

Returns:
    A quantifier-free sentence from the given set of sentences that is not
    satisfied by the given model.
"""
# We assume that every formula in sentences is in prenex normal form and has
# no free variable names, that sentences is universally and existentially
# closed, and that the set of constant names that appear somewhere in
# sentences is model.universe; but we cannot assert these since we cannot
# iterate over sentences.
for constant in model.universe:
    assert is_constant(constant)
assert is_in_prenex_normal_form(unsatisfied)
assert len(unsatisfied.free_variables()) == 0
assert unsatisfied in sentences
assert not model.evaluate_formula(unsatisfied)
# Task 12.2

```

Hint: Use recursion to “peel off” one quantifier at a time (replacing the quantified variable name with some constant name) while maintaining that the resulting sentence is in the given set of sentences and is unsatisfied by the given model. Use the fact that the given set of sentences is universally and existentially closed to guide your implementation and ensure its correctness.

Your solution to Task 2 and the reasoning behind it essentially prove the following lemma (in particular, the reasoning behind your code does not hinge in any way on the finiteness of the set of sentences or constant names.⁵):

Lemma. *Let S be a (possibly infinite) universally and existentially closed set of sentences in prenex normal form. If S is not satisfied by some model M , then there exists a quantifier-free sentence in S that is not satisfied by M .*

So, returning to our proof outline from before Task 2, we now have a quantifier-free sentence that is not satisfied by the model that we created using the primitive closure condition of S . We will notice that this quantifier-free sentence is composed, using logical operators, of only primitive sentences. Generally speaking, the formulas in our language are of course composed of relation invocations and equalities, with operators and quantifications applied to them to create compound formulas. However, having assumed away equalities for the analysis of this chapter, we have that all formulas are composed, using operators and quantifications, solely of relation invocations; and specifically that all quantifier-free sentences are composed, using only operators, of relation invocations applied solely to constant names (since we have assumed away function names, and since any variable name would be free, making the quantifier-free “sentence” in fact not a sentence). In other words, all quantifier-free sentences are composed, using only logical operators, of what we called, exactly for this reason, **primitive** sentences.

⁵While you do iterate over the set of constant names in your solution, you do so only to find a constant name that when substituted into a given sentence yields a sentence that does not hold in the model. The existence of such a constant name is guaranteed by the closure conditions, and this would continue to hold even if the set of constant names were infinite.

To complete our proof we will prove below, as you will demonstrate in the next task, that the quantifier-free sentence from S that is not satisfied by the model that we created (using the primitive closure condition of S) must tautologically contradict the sentences in S that correspond to the primitive sentences from which this quantifier-free sentence is composed, and therefore S is inconsistent.

Task 3. Implement the missing code for the function `model_or_inconsistency(sentences)`, which either returns a model of the given closed (finite) set of sentences in prenex normal form (if such a model exists), or returns a proof of a contradiction from these sentences (as well as our axioms) as assumptions.

```

predicates/completeness.py

def get_primitives(quantifier_free: Formula) -> Set[Formula]:
    """Finds all primitive subformulas of the given quantifier-free formula.

    Parameters:
        quantifier_free: quantifier-free formula that contains no function names
                        and no equalities, whose subformulas are to be searched.

    Returns:
        The primitive subformulas (i.e., relation invocations) of which the
        given quantifier-free formula is composed using logical operators.

    Examples:
        The primitive subformulas of '(R(c1,d)|~(Q(c1)->~R(c2,a)))' are
        'R(c1,d)', 'Q(c1)', and 'R(c2,a)'.
    """
    assert is_quantifier_free(quantifier_free)
    assert len(quantifier_free.functions()) == 0
    assert '=' not in str(quantifier_free)
    # Task 12.3a

def model_or_inconsistency(sentences: AbstractSet[Formula]) -> \
    Union[Model[str], Proof]:
    """Either finds a model in which the given closed set of prenex-normal-form
    sentences holds, or proves a contradiction from these sentences.

    Parameters:
        sentences: closed set of prenex-normal-form sentences that contain no
                function names and no equalities, to either find a model of, or
                prove a contradiction from.

    Returns:
        A model in which all of the given sentences hold if such exists,
        otherwise a valid proof of a contradiction from the given formulas via
        `Prover.AXIOMS`.
    """
    assert is_closed(sentences)
    for sentence in sentences:
        assert len(sentence.functions()) == 0
        assert '=' not in str(sentence)
    # Task 12.3b

```

Guidelines: First construct the model with `get_constants(sentences)` as its universe and with relation interpretations according to the primitive (and negation-of-primitive) sentences in `sentences` (ignoring any sentence in `sentences` that is not a primitive sentence or its negation while constructing this model). If this model satisfies `sentences`,

then you are done. Otherwise, find some sentence from `sentences` that this model does not satisfy, and use Task 2 to consequently find a quantifier-free sentence from `sentences` that this model does not satisfy. Then, tautologically prove a contradiction from 1) this quantifier-free sentence, 2) the primitive sentences in `sentences` that appear in this quantifier-free sentence, and 3) the negation-of-primitive sentences in `sentences` whose primitive negation appears in this quantifier-free sentence. For the last part (proving a contradiction), first implement the missing code for the recursive function `get_primitives(quantifier_free)` (see the function *docstring* for details), and then use that function to complete your solution of this task.

Your solution to Task 3 demonstrates the lemma on Completeness for Closed Sets stated in the beginning of this section. We do not consider your solution to programmatically prove this lemma, though, as the implementation of your solution most probably does not explain *why* a key step of this solution works. Specifically, the last key remaining step in the proof of the lemma on Completeness for Closed Sets, that of deriving a contradiction from the quantifier-free sentence that you found in Task 2, is not necessarily *explained* by your implementation. If you followed our guidelines above, then your implementation states that a contradiction is a tautological implication of this quantifier-free sentence, as well as its primitive components—or their negations—that are in S . Otherwise, you may have done something slightly different but essentially the same such as stating that the negation of the conjunction of this quantifier-free sentence and its primitive components—or their negations—that are in S , is a tautology, and as such simply listed that as a line in the proof of the contradiction. Either way, we will now explicitly, mathematically, prove that a contradiction can in fact be derived as needed:

Lemma. *Let ϕ be a quantifier-free sentence, let p_1, \dots, p_n be its primitive subformulas, and let S be a set of formulas that for every $i = 1, \dots, n$ contains either the primitive sentence p_i or its negation $\neg p_i$. Denote by S' the set of these primitive or negated-primitive sentences that are in S . If ϕ evaluates to *False* in some model of S' , then $S' \cup \{\phi\}$ is inconsistent.*

Proof. Fix a model M of S' in which ϕ does not hold. By definition, M must give the value *True* to every primitive sentence $p_i \in S'$ and must give the value *False* to every primitive sentence $p_i \notin S'$, since in the latter case we have that $\neg p_i \in S'$. Let ψ be the conjunction (i.e., concatenation using ‘&’ operators) of all sentences in S' , i.e., the conjunction of all the p_i sentences/their negations, as each appears in S' . By the semantic definition of evaluating conjunctions, there is a single possible truth-value assignment to all the p_i sentences that makes ψ evaluate to *True*, and this naturally is precisely the above-described truth-value assignment to the p_i sentences by our model M .

We claim that $(\psi \rightarrow \neg \phi)$ is a tautology. To see this, we will view ϕ as a propositional formula over the p_i sentences (to be completely formally, we would have had to phrase the discussion that follows in terms of the propositional skeleton of ϕ). Since ϕ is quantifier-free and p_1, \dots, p_n are *all* of its primitive subformulas, ϕ indeed is a propositional formula over the p_i sentences, that is, it is composed of the p_i sentences using only logical operators. Therefore, the truth value of ϕ in any model is completely determined by the truth values of the p_i sentences in that model, and therefore the truth value of $(\psi \rightarrow \neg \phi)$ in any model is also completely determined by the truth values of the p_i sentences in that model. In order to show that $(\psi \rightarrow \neg \phi)$ is a tautology it therefore suffices to show that any truth-value assignment to the p_i sentences satisfies $(\psi \rightarrow \neg \phi)$. Since, as noted above, there is a single truth-value assignment to the p_i sentences that satisfies ψ , we only need to show that this assignment also satisfies $\neg \phi$, but this exactly is the meaning of our assumption that ϕ gets

value *False* in the model M (which we have argued to correspond to the unique assignment that makes ψ evaluate to *True*).

We can now obtain a proof of a contradiction from $S' \cup \{\phi\}$ by first proving ψ from all of its primitive subformulas/their negations (that are all in S'), and then applying MP to the tautology $(\psi \rightarrow \neg\phi)$ to deduce $\neg\phi$. Together with ϕ we thus prove the contradiction $(\phi \& \neg\phi)$. (Or, concisely as you have probably done in your code, since ψ is a tautological implication of its primitive subformulas/their negations that are in S' , and since MP is also tautological, simply derive $(\phi \& \neg\phi)$, or any other contradiction, as a tautological implication of ϕ as well as all of its primitive subformulas/their negations that are in S' .) \square

Note that the proof of the above lemma uses the fact that we have *every* tautology at our disposal when writing a proof. This is unlike our “usual” usage of tautologies in proofs so far, which was limited to *specific* “nice” or “simple” tautologies that do save some clutter in our proofs but each of which we could have manually proved separately had we wanted to. The fact that we have *every* tautology at our disposal, as really is required in the proof of the above lemma, even if we are not allowed to use tautology lines (as long as we can use the schema equivalents of our propositional axiomatic system instead), is due to the Tautology Theorem that your work in the first part of this book proved. This is a key place on the way to proving the Completeness Theorem for Predicate Logic, where the Tautology Theorem is used.

2 Closing a Set

With the lemma on Completeness for Closed Sets in hand, to deduce the Completeness Theorem it is enough to show that every set of sentences in prenex normal form can be “closed” (i.e., can have sentences added to it to obtain a closed set of sentences in prenex normal form) without losing consistency (i.e., so that if the original set is consistent, then so is the closed set), since then we will be able to first “close” this set and then “read off” a model (or generate a contradiction, if the “original” set of sentences was not consistent to begin with) as in Task 3. It is therefore enough to prove the following lemma:

Lemma (Consistency-Preserving Closure). *For every (possibly infinite) consistent set of sentences in prenex normal form S , there exists a closed consistent superset of sentences in prenex normal form $\bar{S} \supseteq S$.*

The high-level idea of “closing” S is to iteratively add more and more sentences that help satisfy one of the three closure conditions, in a way that does not lose consistency. In the following sequence of tasks you will indeed show that given any nonclosed set of sentences, it is always possible to add to this set an additional sentence that satisfies an additional closure condition, and, crucially, to do this without losing consistency. In the end of this section, we will discuss how to combine your solutions to these various tasks to “close” a set S . This last step will be somewhat involved, and in fact as we will see, for more than one reason we will have no choice but to carry out parts of the proof of this last step mathematically rather than programmatically.

2.1 Primitive Closure

We start addressing the three closure conditions by showing how to satisfy a primitive closure condition. We would like to show that for any primitive sentence ϕ , we can add

either ϕ or its negation ' $\sim\phi$ ' to our consistent set of sentences S without losing consistency. We will actually show more generally that if S is consistent, then for every arbitrary (not necessarily primitive) sentence ϕ , we can add either ϕ or its negation ' $\sim\phi$ ' to S without losing consistency. It is not programmatically easy to figure out which of these can be added without losing consistency, but in the next task you will nonetheless programmatically prove that one of these is possible, by showing that if both $S \cup \{\phi\}$ is inconsistent and $S \cup \{\sim\phi\}$ is inconsistent, then S was already inconsistent to begin with.

Task 4. Implement the missing code for the function `combine_contradictions(proof_from_affirmation, proof_from_negation)`. This function takes as input two proofs of contradictions, both from *almost* the same set of assumption/axiom sentences, with the only difference between the assumptions of the two proofs being that each has an extra simple assumption (with no templates), with the extra assumption of `proof_from_negation` being the negation of the extra assumption of `proof_from_affirmation`. The function returns a proof of a contradiction from the assumptions/axioms that are common to both proofs.

predicates/completeness.py

```
def combine_contradictions(proof_from_affirmation: Proof,
                          proof_from_negation: Proof) -> Proof:
    """Combines the given two proofs of contradictions, both from the same
    assumptions/axioms except that the latter has an extra assumption that is
    the negation of an extra assumption that the former has, into a single proof
    of a contradiction from only the common assumptions/axioms.

    Parameters:
        proof_from_affirmation: valid proof of a contradiction from one or more
            assumptions/axioms that are all sentences and that include
            `Prover.AXIOMS`.
        proof_from_negation: valid proof of a contradiction from the same
            assumptions/axioms of `proof_from_affirmation`, but with one
            simple assumption (i.e., without any templates) `assumption`
            replaced with its negation `~assumption`.

    Returns:
        A valid proof of a contradiction from only the assumptions/axioms common
        to the given proofs (i.e., without `assumption` or its negation).
    """
    assert proof_from_affirmation.is_valid()
    assert proof_from_negation.is_valid()
    common_assumptions = proof_from_affirmation.assumptions.intersection(
        proof_from_negation.assumptions)
    assert len(common_assumptions) == \
        len(proof_from_affirmation.assumptions) - 1
    assert len(common_assumptions) == len(proof_from_negation.assumptions) - 1
    affirmed_assumption = list(proof_from_affirmation.assumptions -
                               common_assumptions)[0]
    negated_assumption = list(proof_from_negation.assumptions -
                              common_assumptions)[0]
    assert len(affirmed_assumption.templates) == 0
    assert len(negated_assumption.templates) == 0
    assert negated_assumption.formula == \
        Formula('~', affirmed_assumption.formula)
    assert proof_from_affirmation.assumptions.issuperset(Prover.AXIOMS)
    assert proof_from_negation.assumptions.issuperset(Prover.AXIOMS)
    for assumption in common_assumptions.union({affirmed_assumption,
```

```

negated_assumption}):
    assert len(assumption.formula.free_variables()) == 0
# Task 12.4

```

Hint: One possible approach is to start by applying the function `proof_by_way_of_contradiction()` to each of the given proofs.⁶

Your solution to Task 4 proves the following lemma:

Lemma. *Let S be a (possibly infinite) consistent set of sentences. For every sentence ϕ , either $S \cup \{\phi\}$ is consistent or $S \cup \{\neg\phi\}$ is consistent (or both).*

Notice that while Task 4 applies to any sentence ϕ , to get the primitive closure condition we only need to apply it to primitive sentences of the form ' $R(c_1, \dots, c_n)$ ' where R is one of the relation names that appear in S and n is the arity of R , and each c_i is a constant name that appears somewhere in S . So, to “primitively close” a finite S , since there are only finitely many such primitive sentences that contain relation names and constant names from S , we could simply go over all of them one by one, adding each of them or its negation (whichever does not violate consistency at that point) until we reach a state in which for each sentence of this form, either it or its negation is in S , i.e., a state in which S is primitively closed. As noted above, it is not programmatically easy to figure out which of these two sentences (the primitive or its negation) can be added without losing consistency,⁷ so we will not program this step of the construction, but will rather make do for the mathematical proof ahead with the knowledge that one of these can indeed be done, as you have proved in Task 4.

2.2 Universal Closure

The next two tasks deal with satisfying a universal closure condition. We would like to show that for any universal sentence ' $\forall x[\phi(x)]$ ' that our consistent set of sentences S contains, we can add to S any universal instantiation ' $\phi(c)$ ' of it, for any constant name c , without losing consistency. To prove this, we will show that if the enlarged set is inconsistent, then S was already inconsistent to begin with.

Task 5. Implement the missing code for the function `eliminate_universal_instantiation_assumption(proof, universal, constant)`. This function takes as input a proof of a contradiction from a set of assumption/axiom sentences, a sentence `universal` that is an assumption of the given proof of the form ' $\forall x[\phi(x)]$ ' for some variable name x , and a constant name `constant` such that another assumption of the given proof is the universal instantiation ' $\phi(\text{constant})$ ' of `universal` with this constant name. The function returns a proof of a contradiction from the same assumptions/axioms except the universal instantiation ' $\phi(\text{constant})$ '.

⁶While this approach results in an implementation that is conceptually not dissimilar from your implementation of `reduce_assumption()` from Chapter 6, and while the functionalities of these two functions are not dissimilar and each of them crucially relies on a respective deduction theorem, we note that each of these two functions plays a decidedly different role in the proof of the completeness theorem that corresponds to it.

⁷If it were easy, then we would not have any conjectures since we would immediately know for each sentence whether or not it contradicts our axioms, and so all Mathematicians would be out of a job...

predicates/completeness.py

```
def eliminate_universal_instantiation_assumption(proof: Proof,
                                                  universal: Formula,
                                                  constant: str) -> Proof:
    """Converts the given proof of a contradiction, whose assumptions/axioms
    include `universal` and `instantiation`, where the latter is the universal
    instantiation of the former with the constant name `constant`, to a proof
    of a contradiction from the same assumptions without `instantiation`.

    Parameters:
        proof: valid proof of a contradiction from one or more
            assumptions/axioms that are all sentences and that include
            `Prover.AXIOMS`.
        universal: assumption of the given proof that is universally quantified.
        constant: constant name such that the formula `instantiation` obtained
            from the statement quantified by `universal` by replacing all free
            occurrences of the universal quantification variable name by the
            given constant name, is an assumption of the given proof.

    Returns:
        A valid proof of a contradiction from the assumptions/axioms of the
        given proof except `instantiation`.
    """
    assert proof.is_valid()
    assert Schema(universal) in proof.assumptions
    assert universal.root == 'A'
    assert is_constant(constant)
    assert Schema(universal.statement.substitute({universal.variable:
                                                  Term(constant)})) in \
        proof.assumptions
    for assumption in proof.assumptions:
        assert len(assumption.formula.free_variables()) == 0
    # Task 12.5
```

Hint: Once again, one possible approach is to start by applying the function `proof_by_way_of_contradiction()` to the given proof.

Your solution to Task 5 proves the following lemma:

Lemma. *Let S be a (possibly infinite) consistent set of sentences. For every universally quantified sentence $\forall x[\phi(x)] \in S$ and every constant name c , we have that $S \cup \{\phi(c)\}$ is consistent.*

So, to “universally close” a finite S , for each universal sentence in S we could go ahead and start by adding to S , without losing consistency, all of the universal instantiations of this sentence with respect to all constant names that appear somewhere in S . This, however, would not guarantee that the enlarged S is universally closed, since some of the added instantiations may themselves be universally quantified. For example, if we start with $\forall x[\forall y[R(x,y,a,b)]]$, then the set after adding all of the universal instantiations of $\forall x[\forall y[R(x,y,a,b)]]$ will be $\{\forall x[\forall y[R(x,y,a,b)]]\}$, $\forall y[R(a,y,a,b)]$, $\forall y[R(b,y,a,b)]$, which is not universally closed since it is missing universal instantiations of the two added sentences $\forall y[R(a,y,a,b)]$ and $\forall y[R(b,y,a,b)]$. So, we would have to repeat this process to add all of their universal instantiations and so on, until we would reach a state in which S is universally closed. Fortunately, this would only take a finite number of repetitions, which equals the maximum number of universal quantifications at the top level of a single formula in the original S (since in each repetition, a newly added universal instantiation

has one fewer quantifier than the universal sentence from the previous repetition that it instantiates). In the next task you will perform one round of this process.

Task 6. Implement the missing code for the function `universal_closure_step(sentences)`, which returns a superset of the given (finite) set of sentences in prenex normal form, that additionally contains, for every universally quantified sentence $\forall x[\phi(x)]$ in the given set, all of the universal instantiations of this sentence with respect to every constant name that appears somewhere in the given sentences.

predicates/completeness.py

```
def universal_closure_step(sentences: AbstractSet[Formula]) -> Set[Formula]:
    """Augments the given sentences with all universal instantiations of each
    universally quantified sentence from these sentences, with respect to all
    constant names from these sentences.

    Parameters:
        sentences: prenex-normal-form sentences to augment with their universal
        instantiations.

    Returns:
        A set of all of the given sentences, and in addition any formula that
        can be obtained from the statement quantified by any universally
        quantified sentence from the given sentences by replacing all
        occurrences of the quantification variable name with some constant name
        from the given sentences.
    """
    for sentence in sentences:
        assert is_in_prenex_normal_form(sentence) and \
            len(sentence.free_variables()) == 0
    # Task 12.6
```

Example: If we call this function with a set containing only the sentence $\forall x[\forall y[R(x,y,a,b)]]$, then the returned set will contain, in addition to the original sentence, also the two sentences $\forall y[R(a,y,a,b)]$ and $\forall y[R(b,y,a,b)]$, and if we call this function again on this returned set, then the new returned set will contain, in addition to the three above sentences, also the four sentences $R(a,a,a,b)$, $R(a,b,a,b)$, $R(b,a,a,b)$, and $R(b,b,a,b)$ (and if we call this function again on this returned set, then the same set would again be returned).

2.3 Existential Closure

Finally, we get to satisfying an existential closure condition. We would like to show that for any existential sentence $\exists x[\phi(x)]$ that our consistent set of sentences S contains, we can add to S an existential witness of the form $\phi(c)$, where c is a new (not previously in S) “witnessing” constant name,⁸ without losing consistency. Similarly to Task 5, to prove this we will show that if the enlarged set is inconsistent, then S (which does not use this new witnessing constant name at all) was already inconsistent to begin with. To do so, given the proof of a contradiction from all of the sentences including the added existential

⁸Such a witnessing constant name is often called a “Henkin” constant name, after the Jewish-American logician Leon Henkin. While the Completeness Theorem was first proven by the Austrian (and later American) logician Kurt Gödel, after whom it is customarily named **Gödel’s Completeness Theorem**, the simpler strategy for proving this theorem that has become standard in Mathematical Logic courses, and which we also follow, is due to Henkin.

witness ' $\phi(c)$ ', you will start with an intermediate step that replaces all of the occurrences of this new witnessing constant name c , in this proof and in all of its assumptions, with a new variable name. So, for example, if this new variable name is ' zz ', then the assumption ' $\phi(c)$ ' becomes ' $\phi(zz)$ ' after the replacement (both as an assumption and as the formula of any lines in which this assumption is instantiated). As it turns out, the validity of the proof is actually maintained following this replacement: the propositional skeletons of line formulas do not change, so tautologies remain tautologies (and for the same reason, the conclusion of the proof remains a contradiction); any MP application remains valid since the same replacements are made in all three formulas involved; any UG application remains valid since variable names remain variable names and the same replacements are made in both formulas involved; and finally, any formula justified by an assumption/axiom remains a legal instance since we perform the replacement also in the assumption/axiom and in any instantiation map, and since there are no quantifications anywhere over the new variable name, so no instance of it can become illegally bound during any instantiation. Given the proof that results from this replacement, you will then show that a contradiction can be proven from the assumptions of this proof even without the assumption ' $\phi(zz)$ ' (which in this proof replaced the existential witness assumption ' $\phi(c)$ ' of the original proof). In other words (since the witnessing constant name is a new constant name that does not appear in any assumption of the original proof other than ' $\phi(c)$ '), a contradiction can be proven from the assumptions of the original proof even without the existential witness ' $\phi(c)$ ', which is what we set out to show.

Task 7.

- a. Implement the missing code for the function `replace_constant(proof, constant, variable)`, which takes a proof, a constant name that (potentially) appears in the assumptions of the proof and/or in the proof itself, and a variable name that does not appear anywhere in the proof or in the assumptions, and returns a “similar” (and still valid) proof where every occurrence of the given constant name in the assumptions and in the proof is replaced with the given variable name.

predicates/completeness.py

```
def replace_constant(proof: Proof, constant: str, variable: str = 'zz') -> \
    Proof:
    """Replaces all occurrences of the given constant name in the given proof
    with the given variable name.

    Parameters:
        proof: valid proof in which to replace.
        constant: constant name that does not appear as a template constant name
            in any of the assumptions of the given proof.
        variable: variable name that does not appear anywhere in the given proof
            or in its assumptions.

    Returns:
        A valid proof where every occurrence of the given constant name in the
        given proof and in its assumptions is replaced with the given variable
        name.
    """
    assert proof.is_valid()
    assert is_constant(constant)
    assert is_variable(variable)
    for assumption in proof.assumptions:
        assert constant not in assumption.templates
```



```

    assert variable not in assumption.formula.variables()
for line in proof.lines:
    assert variable not in line.formula.variables()
# Task 12.7a

```

Hint: the `substitute()` methods of the classes `Formula` and `Term` may be useful here.

- b. Implement the missing code for the function `eliminate_existential_witness_assumption(proof, existential, constant)`. This function takes as input a proof of a contradiction from a set of assumption/axiom sentences, a sentence `existential` that is an assumption of the given proof of the form $\exists x[\phi(x)]$ for some variable name x , and a constant name `constant` such that another assumption of the given proof is the existential witness $\phi(\text{constant})$ of `existential` with this (witnessing) constant name and such that this constant name does not appear anywhere else in the assumptions/axioms. The function returns a proof of a contradiction from the same assumptions/axioms except the existential witness $\phi(\text{constant})$.

predicates/completeness.py

```

def eliminate_existential_witness_assumption(proof: Proof,
                                              existential: Formula,
                                              constant: str) -> Proof:
    """Converts the given proof of a contradiction, whose assumptions/axioms
    include `existential` and `witness`, where the latter is the existential
    witness of the former with the witnessing constant name `constant`, to a
    proof of a contradiction from the same assumptions without `witness`.

    Parameters:
        proof: valid proof, which does not contain the variable name 'zz' in its
            lines or assumptions, of a contradiction from one or more
            assumptions/axioms that are all sentences and that include
            `Prover.AXIOMS`.
        existential: assumption of the given proof that is existentially
            quantified.
        constant: constant name such that the formula `witness` obtained from
            from the statement quantified by `existential` by replacing all free
            occurrences of the existential quantification variable name by the
            given constant name, is an assumption of the given proof, and such
            that this constant name does not appear in any assumption of the
            given proof except `witness`.

    Returns:
        A valid proof of a contradiction from the assumptions/axioms of the
        given proof except `witness`.
    """
    assert proof.is_valid()
    assert Schema(existential) in proof.assumptions
    assert existential.root == 'E'
    assert is_constant(constant)
    witness = existential.statement.substitute({existential.variable:
                                              Term(constant)})
    assert Schema(witness) in proof.assumptions
    for assumption in proof.assumptions:
        assert len(assumption.formula.free_variables()) == 0
        assert 'zz' not in assumption.formula.variables()

```

```

for assumption in proof.assumptions - {Schema(witness)}:
    assert constant not in assumption.formula.constants()
for line in proof.lines:
    assert 'zz' not in line.formula.variables()
# Task 12.7b

```

Guidelines: In the given proof, replace the given constant name with the new variable name ‘zz’ that you may assume does not appear anywhere in the original proof, use `proof_by_way_of_contradiction()` to prove ‘ $\neg\phi(zz)$ ’ from the assumptions without `witness`, and finally prove a contradiction from this and from ‘ $\exists x[\phi(x)]$ ’.

Your solution to Task 7 (along with the above reasoning of why the replacement in the first part of that task maintains the validity of the proof) proves the following lemma:

Lemma. *Let S be a (possibly infinite) consistent set of sentences. For every existentially quantified sentence ‘ $\exists x[\phi(x)]$ ’ $\in S$ and new constant name c that does not appear anywhere in S , we have that $S \cup \{\phi(c)\}$ is consistent.*

So, to “existentially close” a finite S , for each existential sentence in S we could go ahead and start by adding to S , without losing consistency, an existential witness for this sentence using a new witnessing constant name that does not appear anywhere in S . Once again, since some of the added existential witnesses may themselves be existentially quantified (and so each of these would itself require an existential witness for the set to be existentially closed), we may need to repeat this process a finite number of times (which equals the maximum number of existential quantifications at the top level of a single formula in the original S) until we reach a state in which S is existentially closed. Analogously to Task 6 above, in the next task you will perform one round of this process.

Task 8. Implement the missing code for the function `existential_closure_step(sentences)`, which returns a superset of the given (finite) set of sentences in prenex normal form, that additionally contains, for every existentially quantified sentence ‘ $\exists x[\phi(x)]$ ’ in the given set, an existential witness for this sentence with a new witnessing constant name that does not appear anywhere in S , if a witness for this sentence is not already contained in the given sentences.

predicates/completeness.py

```

def existential_closure_step(sentences: AbstractSet[Formula]) -> Set[Formula]:
    """Augments the given sentences with an existential witness that uses a new
    constant name, for each existentially quantified sentence from these
    sentences for which an existential witness is missing.

    Parameters:
        sentences: prenex-normal-form sentences to augment with any missing
            existential witnesses.

    Returns:
        A set of all of the given sentences, and in addition for every
        existentially quantified sentence from the given sentences, a formula
        obtained from the statement quantified by that sentence by replacing all
        occurrences of the quantification variable name with a new constant name
        obtained by calling `next(fresh_constant_name_generator)`.
    """
    for sentence in sentences:
        assert is_in_prenex_normal_form(sentence) and \

```

```
len(sentence.free_variables()) == 0
# Task 12.8
```

Guidelines: Use `next(fresh_constant_name_generator)` (imported for you from `logic_utils.py`) to generate new constant names. You may assume that the given set of sentences does not contain constant names that you have generated this way.

Hint: The `substitute()` method of class `Formula` may be useful both for checking if an existential witness for a given sentence already exists, and for creating such a witness if one does not already exist.

Example: If we call this function with a set containing only the sentence $\exists x[\exists y[R(x,y)]]$, then the returned set will contain, in addition to the original sentence, also the sentence $\exists y[R(e_1,y)]$, where e_1 stands here for a constant name returned by calling `next(fresh_constant_name_generator)`, and if we call this function again on this returned set, then the new returned set will contain, in addition to the two above sentences, also the sentence $R(e_1,e_2)$, where e_2 stands here for an additional constant name returned by calling `next(fresh_constant_name_generator)` (and if we call this function again on this returned set, then the same set would again be returned).

2.4 “Combined” Closure

Let us recap our proposed strategy for proving the Completeness Theorem for Predicate Logic: given a finite set of formulas F , we can convert each formula in F into a sentence in prenex normal form to obtain an equivalent set S . Now, given this finite S , we can first add each primitive sentence (composed of a relation name that is used in S and constant names that appear somewhere in S) or its negation to S so that S becomes primitively closed. While for a given primitive sentence it is hard to determine whether it or its negation should be added to S without losing consistency (i.e., so that if S were consistent before the addition, it would remain consistent), which is why we have not programmed this step, you have shown in Task 4 that one of the two can be added without losing consistency. Once S is primitively closed, we can then add to S all universal instantiations with respect to constant names that appear anywhere in S of all universal sentences from S , as in Task 6, and repeat this process finitely many times until S becomes universally closed. (You have shown in Task 5 that consistency is not lost while doing so.) Once S is primitively and universally closed, we can then add to S existential witnesses for all existential sentences in S as in Task 8, and repeat this process finitely many times until S is existentially closed. (You have shown in Task 7 that consistency is not lost while doing so.) So—have we arrived at a set S that is closed with respect to C ? Unfortunately, while the resulting set S is indeed existentially closed, it may no longer necessarily still be primitively or universally closed, for two reasons. The first reason, which is easy to fix, is that existentially closing S may have added universal sentences to it that are missing instantiations in S —this can be easily fixed by repeatedly alternating between running your solutions of Tasks 6 and 8, so that sentences that start with sequences of alternating quantifications are properly dealt with. The second, more major reason, is that since in the last step (existentially closing S) we have added more constant names to S , we have thus changed the primitive- and universal-closure conditions, as these require that S contain primitive sentences or their negations, and universal instantiations, that are constructed using any constant names that appear somewhere in S . The natural tendency is to keep iterating and alternate between primitively, universally, and existentially closing S , hoping that a closed S will eventually be reached. If a closed S is indeed reached, then as you have shown in Task 3, we can either:

- “read off” a model of S , which is therefore also a model of the “original” S (since the latter is a subset of the “new” S), and is therefore also a model of F since you have shown in Chapter 11 that the prenex normal form of any formula implies that formula (and since the sentence corresponding to a formula implies that formula via UI), and so by the Soundness Theorem, any model of the prenex-normal-form sentence is also a model of the original formula.

—or—

- prove a contradiction from S and from our six logical axiom schemas, in which case due to Tasks 4, 5, and 7, we can prove a contradiction from the “original” S , in which case we can prove a contradiction from F and from our six logical axiom schemas since you have shown in Chapter 11 that the prenex normal form of any formula is provable from that formula and from our six (or equivalently twenty-two) logical axiom schemas (and since the sentence corresponding to a formula is provable from that formula via UG),⁹ so F is inconsistent.

Recall that all of the above can be done if we indeed eventually reach a closed S . However, as it turns out, reaching a closed S may fail to happen within any finite number of steps. Consider, for example, the consistent set S containing only the sentences ‘ $\forall y[\exists x[\text{GT}(x,y)]]$ ’ and ‘ $\text{SAME}(0,0)$ ’. Since the constant name ‘0’ appears in this set of sentences, in order to universally close S our construction adds an existential sentence ‘ $\exists x[\text{GT}(x,0)]$ ’. Now, to existentially close S we need to add an existential witness for this sentence, say, ‘ $\text{GT}(e1,0)$ ’. Once we have added this existential witness, ‘ $e1$ ’ appears in S , so in order to universally close S again, we need to “go back” and add the sentence ‘ $\exists x[\text{GT}(x,e1)]$ ’ to S , which in turn requires adding a new existential witness ‘ $\text{GT}(e2,e1)$ ’ to S , which in turn requires adding the sentence ‘ $\exists x[\text{GT}(x,e2)]$ ’ to S , which in turn requires adding a new existential witness ‘ $\text{GT}(e3,e2)$ ’ to S , and so on. (And of course, for any pair of two constant names ‘ e_i ’ and ‘ e_j ’ added this way, we would also need to add either ‘ $\text{GT}(e_i,e_j)$ ’ or its negation—whichever does not make S lose consistency, or whichever we want if neither does—to get primitive closure, etc.) In fact, if we happened to start with a few more sentences that force ‘GT’ to be a **strict total order** (and in particular, transitive: ‘ $\forall x[\forall y[\forall z[(((\text{GT}(x,y) \& \text{GT}(y,z)) \rightarrow \text{GT}(x,z))]]]$ ’ and antisymmetric: ‘ $\forall x[\forall y[(\text{GT}(x,y) \rightarrow \neg \text{GT}(y,x))]]$ ’), then no finite model can satisfy these as well as ‘ $\forall y[\exists x[\text{GT}(x,y)]]$ ’ and ‘ $\text{SAME}(0,0)$ ’.¹⁰ So, indeed it is not possible to stop after any finite number of steps of any kind, regardless of their order, as otherwise we would reach a closed S with a finite number of constant names, which would imply the existence of a finite model for the above consistent set of sentences (including transitivity and antisymmetry), while as we explained such a model cannot exist. Programmatically, we will thus not be able to close our set of sentences. Luckily, as we will now see, mathematically we may continue alternating between our three different step types “to infinity” and obtain a closed set. Then we will be able to complete the proof of the Completeness Theorem exactly as you have done in Task 3 and as detailed above: by either “reading off” a model of S from this closed set, which is also a model of F , or proving a contradiction from S ,

⁹And as you have shown in Chapter 9, by the first part of this book this may even be proven without tautology lines, with the schema equivalents of our propositional axiomatic system instead.

¹⁰Intuitively, if we interpret ‘ $\text{GT}(x,y)$ ’ as $x > y$, then ‘ $\forall y[\exists x[\text{GT}(x,y)]]$ ’ implies that for every element there exists an even-greater element, and ‘GT’ being a strict total order means, loosely speaking, that no “cycles” of any length (such as $0 > e2 > e1 > 0$ or even $0 > 0$) are allowed. So, to satisfy all of these, an infinite chain $0 < e1 < e2 < \dots$ is required.

which can be transformed via Tasks 4, 5, and 7, and via your solution of Chapter 11, to a proof of a contradiction from F .

Proof of the lemma on Consistency-Preserving Closure. We start with a consistent set S of sentences in prenex normal form that we enumerate¹¹ as $S = \{s_1, s_2, s_3, \dots\}$. We will assume that we have access to an infinite reservoir of constant names (say, those generated by `next(fresh_constant_name_generator)`) that do not appear in S . We will build a sequence of finite sets $S_0 \subseteq S_1 \subseteq S_2 \subseteq \dots$ (therefore only finitely many constant names and relation names appear in each) such that $S \cup S_i$ is consistent for every i , and such that each S_i 1) contains s_1, \dots, s_i , 2) contains either ' $R(c_1, \dots, c_n)$ ' or ' $\sim R(c_1, \dots, c_n)$ ' for every relation name R that appears somewhere in S_{i-1} and every tuple of constant names (c_1, \dots, c_n) where n is the arity of R and each c_j appears somewhere in S_{i-1} , 3) contains all universal instantiations via constant names that appear somewhere in S_{i-1} of every universal formula in S_{i-1} , and 4) contains an existential witness for every existential formula in S_{i-1} .

As the basis of our construction, we set $S_0 = \emptyset$. At step i , we start with $S_i = S_{i-1}$, and augment it as follows:

1. Add s_i to S_i . (This does not change $S \cup S_i$, so its consistency is maintained.)
2. Sequentially iterating over all (finitely many) relation names R that appear somewhere in S_{i-1} and (finitely many) tuples of constant names (c_1, \dots, c_n) where n is the arity of R and each c_j appears somewhere in S_{i-1} , we in turn add either ' $R(c_1, \dots, c_n)$ ' or its negation ' $\sim R(c_1, \dots, c_n)$ ' to S_i , so that $S \cup S_i$ remains consistent at each point (one of these must maintain consistency by the lemma that corresponds to Task 4).
3. As you have implemented in Task 6, for every universally quantified sentence ' $\forall x[\phi(x)]$ ' $\in S_{i-1}$ and every constant name c that appears somewhere in S_{i-1} , we add the universal instantiation ' $\phi(c)$ ' to S_i (which keeps $S \cup S_i$ consistent by the lemma that corresponds to Task 5).
4. As you have implemented in Task 8, for every existentially quantified sentence ' $\exists x[\phi(x)]$ ' $\in S_{i-1}$ for which an existential witness is not yet present in S_{i-1} , we take a fresh constant name c from our reservoir (so c does not appear anywhere in S and was not used previously), and add the existential witness ' $\phi(c)$ ' to S_i (which keeps $S \cup S_i$ consistent by the lemma that corresponds to Task 7).

Notice that in each step we only add a finite number of sentences and constant names, and therefore at no step is our reservoir of fresh constant names depleted. Now, let us define $\bar{S} = \bigcup_{i=1}^{\infty} S_i$. First note that $S \subseteq \bar{S}$ since for every i , we have that s_i was added to S_i and so $s_i \in S_i \subseteq \bar{S}$. Second, \bar{S} is consistent: otherwise there would have been a proof of a contradiction from \bar{S} , and since this proof would have involved only a finite set of sentences, which would all have thus already been in the same S_i for some (finite) i (since for each k , sentence number k of these finitely many sentences would have been in S_{i_k} for

¹¹As in Propositional Logic, by the way in which we defined predicate-logic formulas, there are only countably many of them since they are represented by finite-length strings over a finite alphabet, so we can enumerate them. If we allowed using constant names or relation names from a set of greater infinite cardinality (e.g., constant names like c_α where α is a real number), then the set of all possible formulas would no longer be countable, and therefore we would not be able to enumerate it. A closed consistent superset of S would still exist, though, and could be constructed via analogous core arguments but with some additional supporting arguments that would require some background in Set Theory.

some i_k , and so all of these finitely many sentences would have been in S_i for $i = \max_k S_{i_k}$, this proof would have shown S_i to be inconsistent, contradicting that by construction it is consistent. Third, \bar{S} is closed: for primitive closure, take some relation name R that appears in \bar{S} and tuple of constant names (c_1, \dots, c_n) where n is the arity of R and each c_j appears somewhere in \bar{S} , then for some i we already have that the relation name R and all c_1, \dots, c_n appear in S_i , so either $'R(c_1, \dots, c_n)'$ or its negation is in $S_{i+1} \subseteq \bar{S}$. Similarly for universal closure, take some universally quantified formula $'\forall x[\phi(x)]' \in \bar{S}$ and constant name c that appears somewhere in \bar{S} , then for some i we already have that $'\forall x[\phi(x)]' \in S_i$ and that c appears somewhere in S_i , so the formula $'\phi(c)'$ is in $S_{i+1} \subseteq \bar{S}$. Finally, for existential closure take some existentially quantified formula $'\exists x[\phi(x)]' \in \bar{S}$, then for some i we already have that $'\exists x[\phi(x)]' \in S_i$, so an existential witness $\phi(c)$ for some constant name c exists in $S_{i+1} \subseteq \bar{S}$. \square

3 The Completeness Theorem

The lemmas on Consistency-Preserving Closure and on Completeness for Closed Sets (along with the Prenex Normal Form Theorem, which allows us restrict our analysis in these lemmas to sentences in prenex normal form) together prove the “hard direction” of the Completeness Theorem for Predicate Logic. Since, as explained in the beginning of this chapter, the “easy direction” of this theorem follows from the Soundness Theorem for Predicate Logic, we have thus completed our proof of the Completeness Theorem for Predicate Logic:

Theorem (The Completeness Theorem for Predicate Logic: “Consistency” Version). *A (possibly infinite) set of predicate-logic formulas has a model if and only if it is consistent.*

Looking at our proof of the Completeness Theorem, and in particular at our proof of the lemma on Consistency-Preserving Closure, focusing on the case of (at most) countably infinite sets of sentences, we notice that we have actually proved a somewhat stronger result: that every finite or countably infinite consistent set of formulas has a (at most) countable model. Since by the Soundness Theorem if a set of formulas has a (even uncountable) model then it is consistent, we get the following quite remarkable semantic theorem named after German mathematician Leopold Löwenheim and Norwegian mathematician Thoralf Skolem:

Theorem (The Löwenheim–Skolem Theorem). *Every finite or countably infinite set of formulas that has a model, also has a (at most) countable model.*

Thus, in Predicate Logic we cannot express, using countably many formulas/schemas, conditions that require uncountability. This may sound strange since, as mentioned in Chapter 10, all of Mathematics can be expressed in Predicate Logic using the (finitely many) ZFC axiom schemas, and in particular these schemas can be used to prove **Cantor’s Theorem**, which states that the real numbers (a subset of the universe of any model of ZFC) are uncountable! How is this possible? This riddle is known as “Skolem’s Paradox.” The solution (which was also given by Skolem, and which explains why this is in fact not a paradox) is that while indeed the Löwenheim–Skolem Theorem ensures a countable model of ZFC, that countability is from the point of view of looking from “our model of ZFC” in which this model was constructed. From the constructed model’s point of view, however, this set of “reals of the constructed countable model” is *not* countable! This may happen since countability of a set is defined as the existence of a 1:1 map from that set to the

integers. Such a map will not exist as a set in the constructed model, even though it does exist in “our math.” (And to make things even more confusing, if from the point of view of that model, you were to create within it an “inner” model of ZFC, then its universe would be countable both from our point of view and from the point of view of the original model, but not from the point of view of the “inner” model, and so forth. . .)

4 The Compactness Theorem and the “Provability” Version of the Completeness Theorem

We will now wish to derive an analogue for Predicate Logic of the Compactness Theorem for Propositional Logic:

Theorem (The Compactness Theorem for Predicate Logic). *A set of predicate-logic formulas has a model if and only if every finite subset of it has a model.*

Recall that in Chapter 6, we used the Compactness Theorem, together with the Completeness Theorem for Finite Sets and with the fact that the syntactic notions of proofs and consistency are inherently finite concepts, to prove the general Completeness Theorem (for possibly infinite sets). Since we have already proven the Completeness Theorem for Predicate Logic for any—even infinite—sets, we can use the reverse order of implication to prove the Compactness Theorem for Predicate Logic. First we will note that it is still the case that since proofs are by definition finite, any proof uses finitely many assumptions, and so if an infinite set of formulas is inconsistent, then a contradiction can already be proven from finitely many of these assumptions:

Lemma. *A set of predicate-logic formulas is consistent if and only if every finite subset of it is consistent.*

We will use the Completeness Theorem for Predicate Logic together with this lemma to prove the Compactness Theorem for Predicate Logic: a set of predicate-logic formulas has a model if and only if it is consistent (by the Completeness Theorem), which is true if and only if every finite subset of it is consistent (by the lemma just stated), which is true of and only if every finite subset of it has a model (by the Completeness Theorem once again). That’s it! An illustration of this argument, as well as, for comparison, the argument that we used in Chapter 6 to prove the Completeness Theorem from the Compactness Theorem, is given in Figure 2. As in Propositional Logic, the Compactness Theorem for Predicate Logic is also very useful even on its own.

Finally, as in Propositional Logic, we will want to rephrase the Completeness Theorem also in a way that talks about provability rather than consistency—a way that is analogous to the Tautology Theorem in giving a converse to our original statement of the Soundness Theorem for Predicate Logic:

Theorem (The Completeness Theorem for Predicate Logic: “Provability” Version). *Let X be the set of our basic six logical axiom schemas. For any set A of predicate-logic formulas and any predicate-logic formula ϕ , it is the case that $A \models \phi$ if and only if $A \cup X \vdash \phi$.*

Proof. We will first show that the theorem holds when ϕ is a sentence. Since ϕ holds in every model that satisfies A , there is no model of A where ‘ $\sim\phi$ ’ holds, so the set $A \cup \{\sim\phi\}$ has no model, and so by the “consistency” version of the Completeness Theorem it is inconsistent, so a contradiction is provable from it and from X . By the theorem on

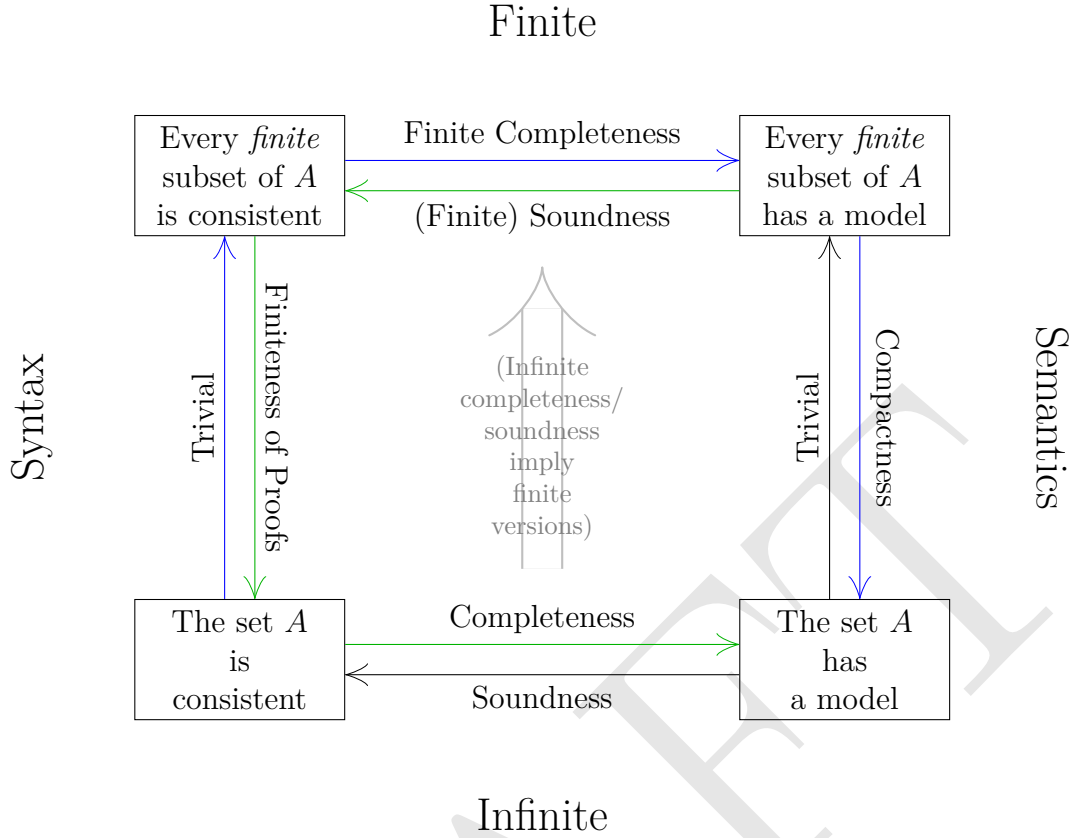


Figure 2: Diagram relating the Completeness and Compactness Theorems in either Propositional Logic or Predicate Logic. The blue arrows trace our proof of (the “hard direction” of) the Completeness Theorem for Propositional Logic in Chapter 6 using Finite Completeness and Compactness. The green arrows trace our proof of (the “hard direction” of) the Compactness Theorem for Predicate Logic in this chapter using the finiteness of proofs and Completeness.

Soundness of Proofs by Way of Contradiction (Task 2 of Chapter 11), which is applicable since ϕ has no free variable names, this means that ‘ $\neg\neg\phi$ ’ is provable from $A \cup X$. Since ϕ is a tautological implication of ‘ $\neg\neg\phi$ ’, we therefore have that ϕ is also provable from $A \cup X$.

For general ϕ , we first universally quantify over all free variable names in ϕ to obtain a sentence ϕ' . Since ϕ holds in every model that satisfies A , so does ϕ' , and so, by the above argument, ϕ' is provable from $A \cup X$. Since ϕ is provable from ϕ' using UI, we therefore have that ϕ is also provable from $A \cup X$. \square

This version of the Completeness Theorem, the crowning achievement of your work throughout this entire book, is, in a sense, justification for the mathematical use of proofs to try and understand what is true in every mathematical structure. Taking A to be the field axioms as in Chapter 10, for instance, this theorem tells us that whatever is true in *every* field can *always* be proven using the field axioms and our six (or equivalently twenty-two) logical axiom schemas. This is quite remarkable: to be fully convinced that a formula holds in every field, instead of (semantically) checking all the infinitely many possible fields, it is always enough to (syntactically) verify the validity of some finite proof! Furthermore, this is true not only for fields, but for any possible axiomatically defined mathematical structure, be it a group, a vector space, the set of natural numbers, or even, as already briefly mentioned in Chapter 10, the entirety of modern Mathematics using the

axiomatization of ZFC (Zermelo–Fraenkel with Choice) Set Theory. Knowing that a proof always exists for any true statement motivates our search for it. Finding the proof (and figuring out what to prove), of course, is an entirely different matter...

DRAFT