Chapter 6:

The Tautology Theorem and the Completeness of Propositional Logic

Recall from Chapter 4 the **Soundness Theorem** for Propositional Logic: any inference rule that is provable (using any set of sound inference rules) is sound. In this chapter, in which our analysis of Propositional Logic culminates, we will prove a converse of sorts: we will show that our proof system, with a specific small axiomatic set of sound inference rules—our **axiomatic system**, which specifically contains **Modus Ponens** as well as few **axioms** (assumptionless inference rules)—is **complete**, that is, can prove every sound inference rule. Our first and main step is to state and prove the **Tautology Theorem**: that every tautology is provable via our axiomatic system. Once we have that under our belt, it will not be difficult to extend this and show that also every sound inference rule is provable, i.e., that if some finite set of assumptions A entails a formula ϕ (i.e., if $A \models \phi$) then ϕ is indeed provable from A using our axiomatic system (i.e., $A \vdash \phi$). Finally, we will further extend this to infinite sets of assumptions, an extension which, due to its infinite nature, we will obviously not prove in a programmatic way but rather using "regular" mathematical reasoning. As we will see in the second part of this book, the Tautology Theorem will also serve as a major building block of our proof of the pinnacle result of this entire book: the Completeness Theorem for the richer Predicate Logic.

1 Our Axiomatic System

In the bulk of this chapter we will restrict ourselves to only allowing the logical operators '→' and '~', disallowing '&', '|', 'T', and 'F'. As we have seen in Exercise 3, these two operators suffice for representing any Boolean function (i.e., synthesizing any truth table), and indeed many Mathematical Logic courses restrict their definitions to only these operators by either disallowing other operators completely, or treating other operators as shorthands for expressions involving only these two operators (along the lines of the substitutions that you implemented in Chapter 3). In the end of the chapter (in Section A) you will optionally see how the same results apply also to formulas that may use the other operators ('|', '&', 'T', 'F', and even others), provided that we add a few appropriate additional axioms to our axiomatic system.

Our axiomatic system—the set of inference rules that we will show to suffice for proving any inference rule—contains all the inference rules that were required for the various tasks of the previous chapter (MP, I0, I1, D, I2, and N), as well as a few additional axioms:

MP: Assumptions: 'p', '(p→q)'; Conclusion: 'q'

I0: '(p→p)'

I1: '(q→(p→q))'

D: ‘((p→(q→r))→((p→q)→(p→r)))’

I2: ‘(~p→(p→q))’

N: ‘((~q→~p)→(p→q))’

NI: ‘(p→(~q→~(p→q)))’

NN: ‘(p→~~p)’

R: ‘((q→p)→((~q→p)→p))’

propositions/axiomatic_systems.py

```
# Axiomatic inference rules that only contain implies

#: Modus ponens / implication elimination
MP = InferenceRule([Formula.parse('p'), Formula.parse('(p->q)')],
                   Formula.parse('q'))
#: Self implication
I0 = InferenceRule([], Formula.parse('(p->p)'))
#: Implication introduction (right)
I1 = InferenceRule([], Formula.parse('(q->(p->q))'))
#: Self-distribution of implication
D = InferenceRule([], Formula.parse('((p->(q->r))->((p->q)->(p->r)))'))

# Axiomatic inference rules for not (and implies)

#: Implication introduction (left)
I2 = InferenceRule([], Formula.parse('(~p->(p->q))'))
#: Converse contraposition
N = InferenceRule([], Formula.parse('((~q->~p)->(p->q))'))
#: Negative-implication introduction
NI = InferenceRule([], Formula.parse('(p->(~q->~(p->q)))'))
#: Double-negation introduction
NN = InferenceRule([], Formula.parse('(p->~~p)'))
#: Resolution
R = InferenceRule([], Formula.parse('((q->p)->((~q->p)->p))'))

#: Large axiomatic system for implication and negation, consisting of `MP`,
#: `I0`, `I1`, `D`, `I2`, `N`, `NI`, `NN`, and `R`.
AXIOMATIC_SYSTEM = frozenset({MP, I0, I1, D, I2, N, NI, NN, R})
```

It is straightforward to verify the soundness of each inference rule in the above axiomatic system, and as already hinted in the previous chapter, your implementation of `is_sound_inference()` has in fact already done so: one of the ways in which we have tested it is by making sure that it successfully verifies the soundness of all of these inference rules.

Whenever we use the notation $\vdash$ in this chapter without explicitly specifying the allowed set $\mathcal{R}$ of inference rules (i.e., whenever we use $\vdash$ rather than $\vdash_{\mathcal{R}}$ for some $\mathcal{R}$), we will mean with respect to the above axiomatic system as the allowed set of inference rules. This set was chosen for ease of use in the tasks of this chapter. Other, smaller, axiomatic systems that still give rise to complete proof systems, are also possible. For example, notice that you have in fact already proven I0 via I1, D, and MP, so certainly by the Lemma Theorem we can remove I0 from the above set without weakening its “proving power.” In fact, it turns out (you will optionally show this in Section B in the end of this chapter) that the subset $\mathcal{H} = \{\mathrm{MP}, \mathrm{I1}, \mathrm{D}, \mathrm{N}\}$ of only four inference rules suggested by the 19th–20th century German mathematician David Hilbert suffices for proving all the others in the above axiomatic system. Thus, once we show that the above axiomatic system can prove something, it follows by the Lemma Theorem that also Hilbert's axiomatic system $\mathcal{H}$ can do so.[1] Thus, even though the tasks ahead allow using the full axiomatic system defined above, we will state our theorems for $\mathcal{H}$.

The full axiomatic system above was chosen as to directly match natural steps in the proofs to come. Specifically, if there is a proof step where we will need to soundly deduce some formula ψ from some other formula ϕ, then we have added to our axiomatic system an axiom of which '(ϕ→ψ)' is a specialization, which will allow us to directly derive ψ from ϕ as required, using an application of MP. If there is a proof step where we will need to soundly deduce some formula ψ from two other formulas ϕ_1 and ϕ_2, then we have added an axiom on which '(ϕ_1→(ϕ_2→ψ))' is a specialization to our axiomatic system. The functions `prove_corollary()` and `combine_proofs()` that you have implemented in the previous chapter will therefore be very useful in applying these axioms—these encoded inference rules—in the corresponding proof steps.

The functions that you are asked to implement in this chapter are contained in the file `propositions/tautology.py` unless otherwise noted.

2 The Tautology Theorem

We now start our journey towards proving a tautology. The first step takes a single model (i.e., a single assignment of truth values to the variable names) and a formula ϕ that evaluates to *True* in this model (a central example being a formula ϕ that is a tautology, i.e., evaluates to *True* in *any* model), and proves the formula ϕ *in this model.* More formally, for each variable name x with value *True* in the given model we take the assumption 'x' and for each variable name x with value *False* in this model we take the assumption '~x'; from these assumptions (one for each variable name used in the formula), we wish to prove the formula ϕ (using the full axiomatic system defined above).

Definition (Formula(s) Capturing Assignment)**.** Given an assignment of a Boolean value b to a variable name x, the formula that **captures** this assignment is the formula 'x' if b is *True* and is the formula '~x' if b is *False*. Given a model $\{x_1 : b_1, x_2 : b_2, \ldots, x_n : b_n\}$, where each x_i is a variable name and each b_i is a Boolean value, the set of formulas that **captures** this model is the set of the n formulas that capture the n assignments in the model.

For example, the model {'p' : *True*, 'q' : *False*, 'r' : *True*} is captured by the set of formulas {'p', '~q', 'r'}. Notice that this definition, while technically trivial, does achieve a transformation from the semantic world of models to the syntactic world of formulas, and as such is an important conceptual step in our proof of the Tautology Theorem. Now suppose that we have a formula ϕ that evaluates to *True* in this model, such as '(p→r)', then our claim is that ϕ is provable, using our axiomatic system, from this set of formulas as assumptions. If, on the other hand, ϕ evaluates to *False* in this model then our claim is that we can prove its negation '~ϕ', using our axiomatic system, from these assumptions. Given such a model and formula ϕ, such a proof (of ϕ or of '~ϕ') can with some care be constructed recursively:

[1]In fact, it is actually possible to derive all of our axioms from the following single axiom, in addition to MP: '(((((p→q)→(~r→~s))→r)→t)→((t→p)→(s→p)))'.

- The base case is very simple since for a variable name x we already have the correct formula (‘x’ if x evaluates to *True*, and ‘$\sim x$’ if x evaluates to *False*) in our set of assumptions.
- If ϕ is of the form ‘$(\phi_1 \to \phi_2)$’ for some formulas ϕ_1 and ϕ_2, then:
  - If ϕ has value *True* in the given model, then either ϕ_1 has value *False* in this model or ϕ_2 has value *True* in it. In the former case we can recursively prove ‘$\sim\phi_1$’, while in the latter we can recursively prove ϕ_2. In the former case, the axiom I2 allows us to use ‘$\sim\phi_1$’ to prove ϕ, and in the latter case the axiom I1 allows us use ϕ_2 to prove ϕ.
  - Otherwise, ϕ has value *False* in the model, so both ϕ_1 is *True* and ϕ_2 is *False* in the model, and so we can recursively prove both ϕ_1 and ‘$\sim\phi_2$’. Now, the axiom NI allows us to prove ‘$\sim\phi$’ from these two.
- Finally, if ϕ is of the form ‘$\sim\psi$’ for some formula ψ, then:
  - If ϕ evaluates to *True*, then ψ evaluates to *False*, so we can recursively prove ‘$\sim\psi$’ which is exactly ϕ, as needed.
  - Otherwise, ϕ evaluates to *False* so ψ evaluates to *True*, so we can recursively prove ψ, but our goal is to prove ‘$\sim\phi$’, that is, ‘$\sim\sim\psi$’, which the axiom NN allows us to prove from ψ.

The fact that the axioms I2, NI, and NN (and I1), which the above proof needs in order to work, are in our axiomatic system is of course no coincidence: we have added these axioms to our axiomatic system precisely to allow us to perform the above steps.

Task 1.

a. First, implement the missing code for the function `formulas_capturing_model(model)`, which returns the formulas that capture the given model.

propositions/tautology.py

```
def formulas_capturing_model(model: Model) -> List[Formula]:
    """Computes the formulas that capture the given model: '`x`' for each
    variable name `x` that is assigned the value ``True`` in the given model,
    and '~`x`' for each variable name `x` that is assigned the value ``False``.

    Parameters:
        model: model to construct the formulas for.

    Returns:
        A list of the constructed formulas, ordered alphabetically by variable
        name.

    Examples:
        >>> formulas_capturing_model({'p2': False, 'p1': True, 'q': True})
        [p1, ~p2, q]
    """
    assert is_model(model)
    # Task 6.1a
```

b. Now, implement the missing code for the function `prove_in_model(formula, model)`, which takes a propositional formula (which may only contain ‘$\to$’ and ‘$\sim$’

as operators) and a model, and returns a proof of either the formula (if it evaluates to *True* in the given model) or its negation (if the given formula evaluates to *False* in the given model) from the formulas that capture the given model as assumptions, via our axiomatic system.

propositions/tautology.py

```
def prove_in_model(formula: Formula, model:Model) -> Proof:
    """Either proves the given formula or proves its negation, from the formulas
    that capture the given model.

    Parameters:
        formula: formula that contains no constants or operators beyond '->' and
            '~', whose affirmation or negation is to prove.
        model: model from whose formulas to prove.

    Returns:
        If the `formula` evaluates to ``True`` in the given model, then a valid
        proof of `formula`; otherwise a valid proof of '~`formula`'. The
        returned proof is from the formulas that capture the given model, in the
        order returned by `formulas_capturing_model(model)`, via
        `AXIOMATIC_SYSTEM`.

    Examples:
        >>> proof = prove_in_model(Formula.parse('(p->q7)'),
        ...                        {'q7': False, 'p': False})
        >>> proof.is_valid()
        True
        >>> proof.statement.conclusion
        (p->q7)
        >>> proof.statement.assumptions
        (~p, ~q7)
        >>> proof.rules == AXIOMATIC_SYSTEM
        True

        >>> proof = prove_in_model(Formula.parse('(p->q7)'),
        ...                        {'q7': False, 'p': True})
        >>> proof.is_valid()
        True
        >>> proof.statement.conclusion
        ~(p->q7)
        >>> proof.statement.assumptions
        (p, ~q7)
        >>> proof.rules == AXIOMATIC_SYSTEM
        True
    """
    assert formula.operators().issubset({'->', '~'})
    assert is_model(model)
    # Task 6.1b
```

Your solution to Task 1 essentially[2] proves the following lemma:

Lemma. *Let ϕ be a formula that only uses the operators '$\rightarrow$' and '$\sim$'. If ϕ evaluates to True in a given model M, then ϕ is provable via $\mathcal{H}$ from the set of formulas that captures M. If ϕ evaluates to False in M, then '$\sim\phi$' is provable via $\mathcal{H}$ from the set of formulas that captures M.*

[2]That is, up to using our full axiomatic system rather than $\mathcal{H}$. See Section B for why $\mathcal{H}$ indeed suffices.

The above lemma implies, in particular, that if ϕ is a tautology, then we can prove it from any set of assumptions that correspond to any model over all of the variable names of ϕ. Our goal, though, is to show that a tautology is provable from no assumptions. We now take an additional step towards this goal by showing how to reduce the number of assumptions by one: how to combine proofs for two models that differ from each other by the value of a single variable name, to eliminate any assumptions regarding that variable name.

Task 2. Implement the missing code for the function `reduce_assumption(proof_from_affirmation, proof_from_negation)`. This functions takes as input two proofs, both proving the *same* conclusion via the same inference rules, from *almost* the same list of assumptions, with the only difference between the assumptions of the two proofs being that the last assumption of `proof_from_negation` is the negation of the last assumption of `proof_from_affirmation`. The function returns a proof of the same conclusion as both proofs, from the assumptions that are common to both proofs (i.e, all assumptions except the last one of each proof), via the same inference rules as well as MP, I0, I1, D, and R.

propositions/tautology.py

```
def reduce_assumption(proof_from_affirmation: Proof,
                      proof_from_negation: Proof) -> Proof:
    """Combines the two given proofs, both of the same formula `conclusion` and
    from the same assumptions except that the last assumption of the latter is
    the negation of that of the former, into a single proof of `conclusion` from
    only the common assumptions.

    Parameters:
        proof_from_affirmation: valid proof of `conclusion` from one or more
            assumptions, the last of which is an assumption `assumption`.
        proof_from_negation: valid proof of `conclusion` from the same
            assumptions and inference rules of `proof_from_affirmation`, but
            with the last assumption being '~`assumption`' instead of
            `assumption`.

    Returns:
        A valid proof of `conclusion` from only the assumptions common to the
        given proofs (i.e., without the last assumption of each), via the same
        inference rules of the given proofs and in addition `MP`, `I0`, `I1`,
        `D`, and `R`.

    Examples:
        If `proof_from_affirmation` is of ``['p', '~q', 'r'] ==> '(p&(r|~r))'``,
        then `proof_from_negation` must be of
        ``['p', '~q', '~r'] ==> '(p&(r|~r))'`` and the returned proof is of
        ``['p', '~q'] ==> '(p&(r|~r))'``.
    """
    assert proof_from_affirmation.is_valid()
    assert proof_from_negation.is_valid()
    assert proof_from_affirmation.statement.conclusion == \
           proof_from_negation.statement.conclusion
    assert len(proof_from_affirmation.statement.assumptions) > 0
    assert len(proof_from_negation.statement.assumptions) > 0
    assert proof_from_affirmation.statement.assumptions[:-1] == \
           proof_from_negation.statement.assumptions[:-1]
    assert Formula('~', proof_from_affirmation.statement.assumptions[-1]) == \
           proof_from_negation.statement.assumptions[-1]
    assert proof_from_affirmation.rules == proof_from_negation.rules
    # Task 6.2
```

Hint: Use the Deduction Theorem (`remove_assumption()` from Chapter 5) and the axiom R.

Once again, the fact that the axiom R, on which your solution to Task 2 relies, is in our axiomatic system is of course no coincidence: we have added this axioms to our axiomatic system precisely because of this. We also note that your solution to Task 2 is the crucial place where the Deduction Theorem is used on the way to proving the Tautology Theorem, as well as the Completeness Theorem for Propositional Logic.

Since Task 1 allows us to prove a tautology ϕ from the set of assumptions that correspond to *any* model, we can keep combining the "combined proofs" resulting from Task 2, each time reducing the number of variable names in the assumptions by one, until we remain with no assumptions. This is precisely how you will now prove the Tautology Theorem.

Task 3 (Programmatic Proof of the Tautology Theorem).

a. Implement the missing code for the function `prove_tautology(tautology, model)`, which returns a proof of the given tautology, via our axiomatic system, from the assumptions that capture the given model, which is a model over a (possibly empty) prefix of the variable names of the given tautology. In particular, if the given model is over the empty set of variable names (the default) then the returned proof is of the given tautology from no assumptions.

propositions/tautology.py

```
def prove_tautology(tautology: Formula, model: Model = frozendict()) -> Proof:
    """Proves the given tautology from the formulas that capture the given
    model.

    Parameters:
        tautology: tautology that contains no constants or operators beyond '->'
            and '~', to prove.
        model: model over a (possibly empty) prefix (with respect to the
            alphabetical order) of the variable names of `tautology`, from whose
            formulas to prove.

    Returns:
        A valid proof of the given tautology from the formulas that capture the
        given model, in the order returned by `formulas_capturing_model(model)`,
        via `AXIOMATIC_SYSTEM`.

    Examples:
        >>> proof = prove_tautology(Formula.parse('(~(p->p)->q)'),
        ...                         {'p': True, 'q': False})
        >>> proof.is_valid()
        True
        >>> proof.statement.conclusion
        (~(p->p)->q)
        >>> proof.statement.assumptions
        (p, ~q)
        >>> proof.rules == AXIOMATIC_SYSTEM
        True

        >>> proof = prove_tautology(Formula.parse('(~(p->p)->q)'))
        >>> proof.is_valid()
        True
        >>> proof.statement.conclusion
        (~(p->p)->q)
        >>> proof.statement.assumptions
        ()
        >>> proof.rules == AXIOMATIC_SYSTEM
        True
    """
    assert is_tautology(tautology)
    assert tautology.operators().issubset({'->', '~'})
    assert is_model(model)
    assert sorted(tautology.variables())[:len(model)] == sorted(model.keys())
    # Task 6.3a
```

Guidelines: If the given model is over all the variable names of the given tautology, simply construct the proof using `prove_in_model(tautology, model)`. Otherwise, recursively call `prove_tautology()` with models that also have assignments to the next variable name that is unassigned in the given model, and then use the `reduce_assumption()` function that you have just implemented.

b. Implement the missing code for the function `proof_or_counterexample(formula)`, which either returns a proof of the given formula from no assumptions via our axiomatic system (if this formula is a tautology) or returns a model in which the given formula does not hold (if this formula is not a tautology).

propositions/tautology.py

```
def proof_or_counterexample(formula: Formula) -> Union[Proof, Model]:
    """Either proves the given formula or finds a model in which it does not
    hold.

    Parameters:
        formula: formula that contains no constants or operators beyond '->' and
            '~', to either prove or find a counterexample for.

    Returns:
        If the given formula is a tautology, then an assumptionless proof of the
        formula via `AXIOMATIC_SYSTEM`, otherwise a model in which the given
        formula does not hold.
    """
    assert formula.operators().issubset({'->', '~'})
    # Task 6.3b
```

You have now shown that every tautology is provable via our (sound) axiomatic system, and since the Soundness Theorem asserts (in particular) that any formula that is provable via any sound set of inference rules is a tautology, you have essentially proven the following theorem, giving a remarkable connection between the semantic and syntactic realms, by stating that any universal truth is provable:

Theorem (The Tautology Theorem). *Every tautology is provable from no assumptions via $\mathcal{H}$. Thus, for any formula ϕ it is the case that $\models \phi$ if and only if $\vdash_{\mathcal{H}} \phi$.*

3 The Completeness Theorem for Finite Sets

The Tautology Theorem directly implies also more general variants of basically the same idea. The first generalization shows that our axiomatic system can be used not only to prove any tautology—that is, any sound inference rule *that has no assumptions*—but in fact to prove any sound inference rule *whatsoever*. Despite its strength, this is in fact a relatively easy corollary as we have already seen in Chapter 5 that one may easily "encode" any inference rule as one without assumptions, such that if the rule is sound then its encoding is a tautology, and such that it is rather easy to get a proof of the rule from a proof of its encoding. In the next task, you will prove this generalization.

Task 4 (Programmatic Proof of the "Provability" Version of the Completeness Theorem for Finite Sets)**.**

a. Start by implementing the missing code for the function `encode_as_formula(rule)`, which returns a single formula that "encodes" the given inference rule.

propositions/tautology.py

```
def encode_as_formula(rule):
    """Encodes the given inference rule as a formula consisting of a chain of
    implications.

    Parameters:
        rule: inference rule to encode.

    Returns:
        The formula encoding the given rule.

    Examples:
        >>> encode_as_formula(InferenceRule([Formula('p1'), Formula('p2'),
        ...                                  Formula('p3'), Formula('p4')],
        ...                                 Formula('q')))
        (p1->(p2->(p3->(p4->q))))

        >>> encode_as_formula(InferenceRule([], Formula('q')))
        q
    """
    # Task 6.4a
```

b. Implement the missing code for the function `prove_sound_rule(rule)`, which takes a sound inference rule, and returns a proof for it via our axiomatic system.

propositions/tautology.py

```
def prove_sound_inference(rule: InferenceRule) -> Proof:
    """Proves the given sound inference rule.

    Parameters:
        rule: sound inference rule whose assumptions and conclusion contain no
            constants or operators beyond '->' and '~', to prove.

    Returns:
        A valid proof of the given sound inference rule via `AXIOMATIC_SYSTEM`.
    """
    assert is_sound_inference(rule)
    for formula in {rule.conclusion}.union(rule.assumptions):
        assert formula.operators().issubset({'->', '~'})
    # Task 6.4b
```

Your solution to Task 4 proves that if a set of assumptions entails a conclusion, then it is indeed possible to prove this conclusion from these assumptions via our (sound) axiomatic system. Thus (similarly to the proof of the Tautology Theorem), together with the Soundness Theorem that gives the converse (for any sound axiomatic system), you have essentially proven the following theorem, expanding our understanding of the connection between the semantic and syntactic realms that we unearthed with the Tautology Theorem:

Theorem (The Completeness Theorem for Finite Sets: "Provability" Version)**.** *For any finite set of formulas A and any formula ϕ, it is the case that $A \models \phi$ if and only if $A \vdash_{\mathcal{H}} \phi$.*

A somewhat more minimalistic and symmetric way to look at the Completeness Theorem is to "move" the conclusion ϕ into the set of assumptions. More specifically, note that by definition, $A \models \phi$ is equivalent to $A \cup \{`{\sim}\phi'\}$ not having a model, and by the theorem on Soundness of Proofs by Way of Contradiction, $A \vdash \phi$ is equivalent to $A \cup \{`{\sim}\phi'\}$ being inconsistent. This allows us to rephrase the Completeness Theorem as an equivalence between the semantic notion of a set of formulas (analogous to $A \cup \{`{\sim}\phi'\}$, but note that '~ϕ' no longer plays a different role here than any of the formulas in A!) not having a model, and the syntactic notion of the same set of formulas being inconsistent.

Task 5 (Programmatic Proof of the "Consistency" Version of the Completeness Theorem for Finite Sets)**.** Implement the missing code for the function `model_or_inconsistency(formulas)`, which either returns a model of the given formulas (if such a model exists), or returns a proof of '~(p→p)' via our axiomatic system from the given formulas as assumptions (if such a model does not exist).

propositions/tautology.py
```
def model_or_inconsistency(formulas: Sequence[Formula]) -> Union[Model, Proof]:
    """Either finds a model in which all the given formulas hold, or proves
    '~(p->p)' from these formulas.

    Parameters:
        formulas: formulas that use only the operators '->' and '~', to either
            find a model of, or prove '~(p->p)' from.

    Returns:
        A model in which all of the given formulas hold if such exists,
        otherwise a valid proof of '~(p->p)' from the given formulas via
        `AXIOMATIC_SYSTEM`.
    """
    for formula in formulas:
        assert formula.operators().issubset({'->', '~'})
    # Task 6.5
```

Hint: If a model of the given formulas does not exist, what can you say about the inference rule with the given formulas as assumptions, and with conclusion '~(p→p)'? Is it sound? Why? Now how can you use your solution to Task 4 to complete your solution? Make sure that you completely understand why your solution works, and do not just be content with it passing our tests.

Your solution for Task 5 indeed proves a version of the Completeness Theorem that is phrased in a more symmetric manner using the semantic notion of having a model and the syntactic notion of consistency:

Theorem (The Completeness Theorem for Finite Sets: "Consistency" Version)**.** *A finite set of formulas has a model if and only if it is consistent with respect to $\mathcal{H}$.*

4 The Compactness Theorem and the Completeness Theorem for Infinite Sets

The theorems we have proven so far in this chapter only proved equivalences between syntactic and semantic notions for *finite* sets of formulas. For example, our last version of the Completeness Theorem stated that a *finite* consistent set of formulas has a model. Does an infinite consistent set of formulas necessarily also have a model? Before providing an answer to this question, let us examine the syntactic definitions of proofs and consistency, and the semantic notions of truth and having a model, for infinite sets of formulas/assumptions.

The definition of a formula ϕ being provable from a set A of assumptions is that a proof exists, and by definition a proof has a *finite* number of lines in it (much like in our discussion of the length of a formula toward the end of Chapter 1, the number of lines of a proof is finite yet unbounded).[3] If the set A is infinite, then any of its infinitely many formulas can be used as an assumption in the proof, but since the proof has finite length, only some finite subset of the assumptions will actually be used in any given proof. Therefore, since inconsistency is defined as having a proof of both a formula and its negation, we have that if an infinite set A is inconsistent, then some finite subset of A (the formulas that appear in the finite proof of the formula and/or in the finite proof of its negation) is already inconsistent. This shows the following trivial lemma:

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

Thus there is really "nothing new" when dealing with infinite sets in terms of the syntactic notions of proofs and consistency—everything can be reduced to finite sets.

Let us now consider the semantic notion of an infinite set A of formulas having a model. The definition requires that the model satisfy every formula in the infinite set. While every formula is, by definition, finite, and thus uses only finitely many variable names, since there are infinitely many formulas in the set, there may be infinitely many variable names that are used altogether in all formulas in the set, so our model will have to be an infinite object: an assignment of a truth value to every variable name that appears in any formula in the set. Determining whether the model satisfies any given formula is still a finite question since any single formula uses only finitely many of the variable names in the model, but the whole notion of an infinite model is new. In particular, so far we have not developed any method of constructing infinite models, so it is not clear how we can construct a model that satisfies an infinite set of formulas.

Surprisingly, it turns out that the question of whether an infinite set of formulas has a model can also be reduced to whether all of its finite subsets have models. For reasons that we will touch on below, this result is called the **Compactness Theorem** for Propositional Logic:

Theorem (The Compactness Theorem for Propositional Logic)**.** *A set of propositional formulas has a model if and only if every finite subset of it has a model.*

[3]Indeed, trying to define any alternative notion of "proofs of infinite length" would completely circumvent the most fundamental goal of Mathematical Logic: understanding what is possible for humans to prove.

The "only if" direction of this theorem is trivial: if the set A has a model, then this model is a model of every subset of A (finite or otherwise). The "if" direction of this theorem is actually quite surprising. Let us look at an analogous statement with "formulas" over real numbers. Consider the set of inequalities $\{x \geq 1, x \geq 2, x \geq 3, \ldots\}$. Clearly every finite subset of these inequalities has a model, i.e., a real number x that satisfies every inequality in this finite subset (say, take the maximum of all of the numbers that are "mentioned" in this finite subset). However, no model, i.e., no value of x, satisfies *all* of these infinitely many inequalities (the set of natural numbers contains no maximum element. . .). In Mathematics, and specifically in the branch of Mathematics called **Topology**, the property that an infinite set of constraints is guaranteed to have a "solution" (a.k.a. a model, a.k.a. nonempty intersection) if every finite subset of it has one, is called **compactness**. Thus, the Compactness Theorem for Propositional Logic is so named since it asserts that for any family of variable names, the space of models over these variable names is **compact** with respect to the constraints of satisfying formulas, and this theorem is in fact a special case one of the cornerstone theorems of Topology, **Tychonoff's Theorem**, which asserts that all of the spaces in a very large family of spaces (including, for any family of variable names, the space of all models over these variable names, and also including many other spaces) have this property.[4] We will now give a direct proof of the Compactness Theorem for Propositional Logic that does not require any background in Topology:

Proof of the Compactness Theorem for Propositional Logic. We prove the "if" direction, that is, we assume that every finite subset of a set F of formulas has a model, and we will construct a model of F. Let us enumerate the variable names used anywhere in the set of formulas as $x_1, x_2, \ldots$, and let us enumerate the formulas in the set as $\phi_1, \phi_2, \ldots$. (There are only countably many of each, so we can indeed enumerate them.[5]) A model is an assignment of a truth value to each of the (possibly infinitely many) variable names. We will build the desired model step by step, at each step i deciding which truth value to assign to x_i. Our goal is to pick the truth value to assign to x_i at the ith step to maintain the following property: for every finite subset of the formulas in F, there exists a model

[4]A remark for readers with a background in Topology: viewing the set of all models as the set of Boolean assignments to countably many variable names, $\{\mathit{True}, \mathit{False}\}^{\mathbb{N}}$, it turns out that the set of models satisfying a given formula is **closed** with respect to the **product topology** over $\{\mathit{True}, \mathit{False}\}^{\mathbb{N}}$, since belonging to this set depends only on finitely many variable name assignments. So, since Tychonoff's Theorem asserts that the product of any collection of compact topological spaces is compact with respect to the product topology, we have as a special case that $\{\mathit{True}, \mathit{False}\}^{\mathbb{N}}$ is compact with respect to the product topology, and so its closed sets satisfy the **finite intersection property**: any collection of closed sets with an empty intersection has a finite subcollection with an empty intersection, which proves the Compactness Theorem for Propositional Logic. Unsurprisingly, the direct proof that we will give for the Compactness Theorem for Propositional Logic is in fact quite similar to a popular proof of the special case of Tychonoff's Theorem for countable products of general metric spaces (the latter proof proves that the product space is sequentially compact, which suffices since the product of countably many metric spaces is metrizable, and for metrizable spaces sequential compactness is equivalent to compactness).

[5]As discussed in Chapter 1, according to our definitions the set of variable names is countably infinite and therefore so is the set of formulas, so we can enumerate each of them. If we allowed using variable names from a set of greater infinite cardinality (e.g., constant names like c_α where α is a real number, which are impossible to enumerate), then the set of all possible formulas would also no longer be countable, and therefore we would not be able to enumerate either of these sets. The Compactness Theorem would still hold, though, and could be proven via analogous core arguments but with some additional supporting arguments that would require some background in Set Theory. Alternatively, even for variable name sets of arbitrary cardinality, the Compactness Theorem for Propositional Logic is still a special case of, and therefore implied by, Tychonoff's Theorem.

that satisfies all of them and has the already-fixed truth values for $x_1, \ldots, x_i$.

Clearly, at the beginning before the first step, the above property holds as we have not fixed any truth values yet, so the assumption of the theorem is exactly this property for $i = 0$. Now let us assume that we already fixed truth values for $x_1, \ldots, x_{i-1}$ with the above property for $i-1$, and see that we can also fix a truth value for x_i in a way that maintains this property for i. For each j let us examine the finite set of formulas $F_j = \{\phi_1, \phi_2, \ldots, \phi_j\}$. Given the above property for $i-1$, for every j there exists a model of F_j with the already-fixed truth values of $x_1, \ldots, x_{i-1}$, so let us consider such a model for each j. Each of these models (that correspond to different values of j) assigns some truth value to x_i, and since there are only two possible truth values, one (at least) of these two values is assigned to x_i by infinitely many of these models. We will pick such a value as our fixed value for x_i. To see that we have the required property for i, consider any finite set $G \subseteq F$. Since G is finite, it is contained in some F_k, and since the value that we fixed for x_i is assigned to it by models corresponding to infinitely many values of j, there exists some $j \geq k$ such that the model that satisfies all of F_j assigns this truth value to x_i (and also has the already-fixed truth values for $x_1, \ldots, x_{i-1}$ since we only looked at models with these fixed values). Since $G \subseteq F_k \subseteq F_j$, this model satisfies G as required and therefore the above property holds also for i.

We have finished describing the construction of the model—a construction that assigned a truth value to every variable name x_i. To complete the proof, we need to show that this model indeed satisfies all of F. Take any formula $\phi \in F$, and let n be the largest index of a variable name that appears in it. The truth value of ϕ depends only on the truth values of the variable names $x_1, \ldots, x_n$, and since we picked the n fixed truth values of these variable names in a way that guarantees that there exists a model with these truth values that satisfies any finite subset of F, and in particular satisfies the singleton set $\{\phi\}$, we conclude that all models that have these fixed values for $x_1, \ldots, x_n$ satisfy ϕ, including the model that we constructed. □

We are now in an enviable state where both the syntactic question of being consistent and the semantic question of having a model can be reduced to finite sets, for which we have already proven the equivalence between the two, so we can now get the same equivalence for infinite sets as well:

Theorem (The Completeness Theorem for Propositional Logic: "Consistency" Version)**.** *A set of formulas has a model if and only if it is consistent with respect to $\mathcal{H}$.*

Proof. By the Compactness Theorem, a set of formulas has a model if and only if every finite subset of it has a model, which by the "consistency" version of the Completeness Theorem for Finite Sets holds if and only if every finite subset of it is consistent, which by an immediate lemma we have proven in the beginning of this section holds if and only if the entire set is consistent. An illustration of this argument is given in Figure 1. □

Again, this formalism of the **Completeness Theorem** is directly equivalent to the other formalism, completing our understanding of the equivalence within Propositional Logic of the semantic notion of entailment and the syntactic notion of provability:

Theorem (The Completeness Theorem for Propositional Logic: "Provability" Version)**.** *For any set of formulas A and any formula ϕ, it is the case that $A \models \phi$ if and only if $A \vdash_{\mathcal{H}} \phi$.*

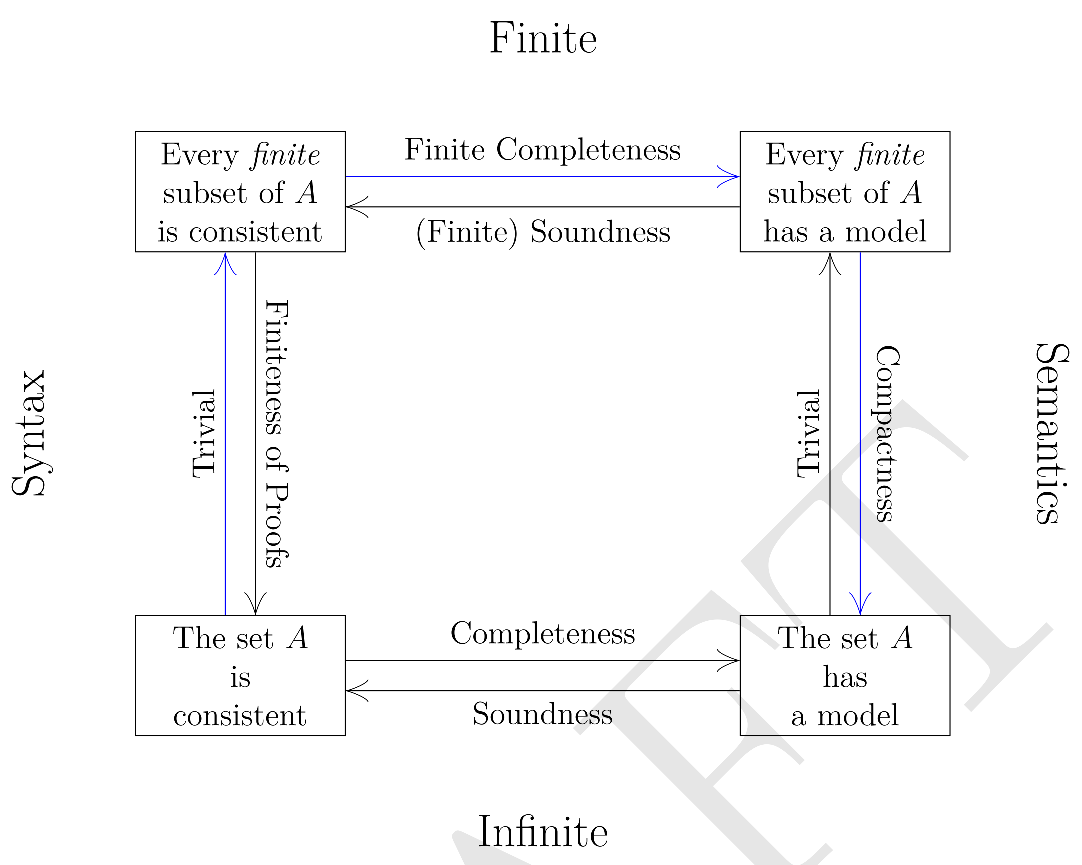

Figure 1: Diagram relating the Completeness and Compactness Theorems. The blue arrows trace our proof of the "hard direction" of the Completeness Theorem for Propositional Logic using Finite Completeness and Compactness.

Proof. $A \models \phi$ is by definition equivalent to $A \cup \{\sim\phi\}$ not having a model, and by the theorem on Soundness of Proofs by Way Of Contradiction, $A \vdash \phi$ is equivalent to $A \cup \{\sim\phi\}$ being inconsistent (nothing in the proof of that theorem depended on the finiteness of A). Therefore, this version of the Completeness Theorem follows immediately from its "consistency" version. □

A Optional Reading: Adding Additional Operators

Throughout this chapter we limited ourselves to formulas that only used negation ('~') and implication ('→'). As we have seen in Chapter 3 that these two operators form a complete set of operators, we have not lost any proving power by limiting ourselves this way. But what if we want to consider general formulas that may also contain the additional operators '&', '|', 'T', and 'F'? One way to treat this question is to view these operators as simply shorthand notation for a full expression that is always only written using negation and implication (along the lines of the substitutions that you implemented in Chapter 3). Under this point of view, it is clear that everything that we proved in this chapter continues to hold, since the new operators are only "syntactic sugar" while the "real" formulas continue to be of the form discussed.

A more direct approach would nonetheless want to actually consider general formulas that may contain '&', '|', 'T', and 'F' as primitive operators, as we have done up to this chapter.[6] It is clear that without further axioms we will not be able to prove anything about a formula like '(p|~p)'. We will thus have to add to our axiomatic systems additional (sound) axioms that essentially capture the properties of these additional operators:

A: '(p→(q→(p&q)))'

NA1: '(~q→~(p&q))'

NA2: '(~p→~(p&q))'

O1: '(q→(p|q))'

O2: '(p→(p|q))'

NO: '(~p→(~q→~(p|q)))'

T: 'T'

NF: '~F'

propositions/axiomatic_systems.py

```
# Axiomatic inference rules for conjunction (and implication and negation)

#: Conjunction introduction
A = InferenceRule([], Formula.parse('(p->(q->(p&q)))'))
#: Negative-conjunction introduction (right)
NA1 = InferenceRule([], Formula.parse('(~q->~(p&q))'))
#: Negative-conjunction introduction (left)
NA2 = InferenceRule([], Formula.parse('(~p->~(p&q))'))

# Axiomatic inference rules for disjunction (and implication and negation)

# Disjunction introduction (right)
O1 = InferenceRule([], Formula.parse('(q->(p|q))'))
# Disjunction introduction (left)
O2 = InferenceRule([], Formula.parse('(p->(p|q))'))
# Negative-disjunction introduction
NO = InferenceRule([], Formula.parse('(~p->(~q->~(p|q)))'))

# Axiomatic inference rules for constants (and implication and negation)

#: Truth introduction
T =  InferenceRule([], Formula.parse('T'))
#: Negative-falsity introduction
NF = InferenceRule([], Formula.parse('~F'))

#: Large axiomatic system for all operators, consisting of the rules in
#: `AXIOMATIC_SYSTEM`, as well as `A`, `NA1`, `NA2`, `O1`, `O2`, `NO`, `T`, and
#: `NF`.
AXIOMATIC_SYSTEM_FULL = \
    frozenset(AXIOMATIC_SYSTEM.union({A, NA1, NA2, O1, O2, NO, T, NF}))
```

[6]One may similarly want to handle as primitives even more operators as introduced in Chapter 3. While this can be similarly done, we will stick to these four operators for the purposes of this section.

So our augmented axiomatic system contains all together seventeen rules: the original nine rules, as well as these additional eight rules. As in the beginning of this chapter, these additional axioms were chosen not for frugality in the number of axioms or for aesthetic reasons, but rather for ease of use. Indeed, once again it turns out (you will optionally show this in Section B) that a smaller and in a sense also more aesthetic subset $\hat{\mathcal{H}}$ of only twelve inference rules suffices for proving all the others in the above augmented axiomatic system, so once again even though Optional Task 6 below allows using the full augmented axiomatic system defined above, the Lemma Theorem allows us to state the corresponding theorem for $\hat{\mathcal{H}}$.

Essentially the only place where something has to be augmented in our analysis so far (and in your already-coded solution to the previous tasks of this chapter) to support these additional operators is the first lemma on the way to the Tautology Theorem—the one proving a formula from assumptions that capture a single model (corresponding to the function `prove_in_model()` that you have implemented in your solution to Task 1). That proof proceeded by induction (recursion) on the formula structure and was tailored closely to the two operators that we allowed to appear there. The same induction/recursion however can be very similarly extended to also handle the additional operators using the above new axioms.

Optional Task 6. Implement the missing code[7] for the function `prove_in_model_full(formula, model)`, which has the same functionality as `prove_in_model(formula, model)` except that the given formula may contain also the operators '&', '|', 'T', and 'F', and the returned proof is via the seventeen inference rules in `AXIOMATIC_SYSTEM_FULL`.

propositions/tautology.py

```
def prove_in_model_full(formula: Formula, model: Model) -> Proof:
    """Either proves the given formula or proves its negation, from the formulas
    that capture the given model.

    Parameters:
        formula: formula that contains no operators beyond '->', '~', '&', and
            '|' (and may contain constants), whose affirmation or negation is to
            prove.
        model: model from whose formulas to prove.

    Returns:
        If `formula` evaluates to ``True`` in the given model, then a valid
        proof of `formula`; otherwise a valid proof of '~`formula`'. The
        returned proof is from the formulas that capture the given model, in the
        order returned by `formulas_capturing_model(model)`, via
        `AXIOMATIC_SYSTEM_FULL`.

    Examples:
        >>> proof = prove_in_model_full(Formula.parse('(p&q7)'),
        ...                             {'q7': True, 'p': True})
        >>> proof.is_valid()
        True
        >>> proof.statement.conclusion
        (p&q7)
        >>> proof.statement.assumptions
        (p, q7)
```

[7]Start by copying the code of your implementation of `prove_in_model()`, and then extend it.

```
        >>> proof.rules == AXIOMATIC_SYSTEM_FULL
        True

        >>> proof = prove_in_model_full(Formula.parse('(p&q7)'),
        ...                             {'q7': False, 'p': True})
        >>> proof.is_valid()
        True
        >>> proof.statement.conclusion
        ~(p&q7)
        >>> proof.statement.assumptions
        (p, ~q7)
        >>> proof.rules == AXIOMATIC_SYSTEM_FULL
        True
    """
    assert formula.operators().issubset({'T', 'F', '->', '~', '&', '|'})
    assert is_model(model)
    # Optional Task 6.6
```

Your solution to Optional Task 6 gives us a general version of the corresponding lemma, this time for formulas that may also use the operators *and*, *or*, *T*, and *F*, in addition to *not* and *implies*:

Lemma. *Let ϕ be a formula (that may use any of the operators '~', '→', '|', '&', 'T', and 'F'). If ϕ evaluates to True in a given model M, then ϕ is provable via $\hat{\mathcal{H}}$ from the set of formulas that captures M. If ϕ evaluates to False in M, then '~ϕ' is provable via $\hat{\mathcal{H}}$ from the set of formulas that captures M.*

Once you have solved Optional Task 6 and have thus proved the above generalized lemma, you could now replace every call to `prove_in_model()` in your code with a call to `prove_in_model_full()` (and replace every reference to the earlier version of the lemma in our analysis with a reference to the generalized version) and all your code should then run[8] (and all proofs should then hold) for general formulas, with our augmented axiomatic system in lieu of the old one.

B Optional Reading: Other Axiomatic Systems

As noted throughout this chapter, we have chosen the axiomatic systems so far to allow for our proofs and code to be as simple as possible. In this section, we will explore other axiomatic systems that still give rise to a complete proof system that are popular due to their small size or due to other aesthetic reasons. Our first order of business is of course to show that the axiomatic system $\mathcal{H} = \{\mathrm{MP}, \mathrm{I1}, \mathrm{D}, \mathrm{N}\}$ still gives rise to a complete proof system (if our formulas may contain only *implies* and *not*).

propositions/axiomatic_systems.py

```
#: Hilbert axiomatic system for implication and negation, consisting of `MP`,
#: `I1`, `D`, and `N`.
HILBERT_AXIOMATIC_SYSTEM = frozenset({MP, I1, D, N})
```

To prove that $\mathcal{H}$ gives rise to a complete proof system, by the Lemma Theorem it suffices to prove via only $\mathcal{H}$ all of the other axioms of the axiomatic system that we have already proven to give rise to a complete proof system, i.e., I0, I2, NI, NN, and R. As

[8]Some assertions would also have to be changed for this to actually run.

you have already proven I0 via (a subset of) $\mathcal{H}$ in Chapter 4, it in fact suffices to prove I2, NI, NN, and R, via MP, I0, I1, D, and N. The Deduction Theorem, both directly (by its explicit use) and indirectly (by using hypothetical syllogisms, via your implementation of `prove_hypothetical_syllogism()` from Chapter 5) will be instrumental in proving these. The functions that you are asked to implement in this section are contained in the file in the file `propositions/some_proofs.py`.

Optional Task 7. Prove the rules I2, NI, NN, and R, each via (a subset of your choosing of) MP, I0, I1, D, and N. The proofs should be respectively returned by the functions `prove_I2()`, `prove_NI()`, `prove_NN()`, and `prove_R()`, whose missing code you should implement.

`propositions/some_proofs.py`

```
def prove_I2() -> Proof:
    """Proves `I2` via `MP`, `I0`, `I1`, `D`, `N`.

    Returns:
        A valid proof of `I2` via the inference rules `MP`, `I0`, `I1`, `D`, and
        `N`.
    """
    # Optional Task 6.7a
       ⋮
def prove_NN() -> Proof:
    """Proves `NN` via `MP`, `I0`, `I1`, `D`, `N`.

    Returns:
        A valid proof of `NN` via the inference rules `MP`, `I0`, `I1`, `D`, and
        `N`.
    """
    # Optional Task 6.7c
       ⋮
def prove_NI() -> Proof:
    """Proves `NI` via `MP`, `I0`, `I1`, `D`, `N`.

    Returns:
        A valid proof of `NI` via the inference rules `MP`, `I0`, `I1`, `D`, and
        `N`.
    """
    # Optional Task 6.7e
       ⋮
def prove_R() -> Proof:
    """Proves `R` via `MP`, `I0`, `I1`, `D`, `N`.

    Returns:
        A valid proof of `R` via the inference rules `MP`, `I0`, `I1`, `D`, and
        `N`.
    """
    # Optional Task 6.7g
```

Guidelines: Use the following strategy:

a. Prove I2 (implement the missing code for `prove_I2()`).

   Hint: Use a hypothetical syllogism (see Task 5 in Chapter 5) whose assumptions are specializations of I1 and N.

b. Prove that '(~~p→p)' (implement the missing code for the function `_prove_NNE()` in the same file).

   Hint: Use the Deduction Theorem, i.e., assume '~~p' and deduce 'p'. To do so, first prove that '(~~p→(~~p→p))' by "chaining" two hypothetical syllogisms (so the conclusion of the first hypothetical syllogism serves as the first assumption of the second hypothetical syllogism). As the first assumption for the first hypothetical syllogism, use '(~~p→(~~~~p→~~p))' (what is this a specialization of?), and as all other assumptions use specializations of N.

c. Prove NN (implement the missing code for `prove_NN()`).

   Hint: Apply MP to an appropriate specialization of N.

d. Prove that '((p→q)→(~q→~p))' (implement the missing code for the function `_prove_CP()` in the same file).

   Hint: First use the Deduction Theorem (twice) to prove '((p→q)→(~~p→~~q))'.

e. Prove NI (implement the missing code for `prove_NI()`).

   Hint: Use a specialization of the assumptionless rule that you have proved in the previous step of this task to prove '(~q→~(p→q))' from 'p'.

f. Prove the inference rule with assumption '(~p→p)' and conclusion 'p' (implement the missing code for the function `_prove_CM()` in the same file).

   Hint: Start your proof with a single line whose conclusion is the formula '((~p→(p→~(~p→p)))→((~p→p)→(~p→~(~p→p))))'.

g. Prove R (implement the missing code for `prove_R()`).

   Hint: Use the Deduction Theorem twice, i.e., prove the inference rule with assumptions '(q→p)', '(~q→p)' and conclusion 'p'. To do so, use a hypothetical syllogism whose assumptions are '(~p→~q)' and '(~q→p)'.

While $\mathcal{H}$ is a very popular axiomatic system due to its relative small size, it is by far not the only axiomatic system of its size that gives rise to a complete proof system. Just to give one example, another popular choice for such an axiomatic system is $\mathcal{H}' = \{\text{MP}, \text{I1}, \text{D}, \text{N'}\}$, the variant of $\mathcal{H}$ obtained by replacing N with the following (sound) axiom as the axiom that captures the properties of the operator '~':

N': '((~q→~p)→((~q→p)→q))'.

propositions/axiomatic_systems.py
```
# Alternative for N

#: Reductio ad absurdum
N_ALTERNATIVE = InferenceRule([], Formula.parse('((~q->~p)->((~q->p)->q))'))

#: Hilbert axiomatic system for implication and negation, with `N` replaced by
#: `N_ALTERNATIVE`.
HILBERT_AXIOMATIC_SYSTEM_ALTERNATIVE = frozenset({MP, I1, D, N_ALTERNATIVE})
```

Once again, to prove that $\mathcal{H}'$ gives rise to a complete proof system, by the Lemma Theorem it suffices to prove that N via (only) MP, I0, I1, D, and N'.

Optional Task 8. Prove the rule N via MP, I0, I1, D, and N'. The proof should be returned by the function `prove_N()`, whose missing code you should implement.

propositions/some_proofs.py
```
def prove_N() -> Proof:
    """Proves `N` via `MP`, `I0`, `I1`, `D`, and `N_ALTERNATIVE`.

    Returns:
        A valid proof of `N` via the inference rules `MP`, `I0`, `I1`, `D`, and
        `N_ALTERNATIVE`.
    """
    # Optional Task 6.8
```

Hint: Use the Deduction Theorem, i.e., assume '(~q→~p)' and prove '(p→q)'. To do so, apply MP to N' to obtain a formula that will serve as the second assumption of a hypothetical syllogism whose conclusion is '(p→q)'.

Axiomatic systems are sometimes chosen not only due to their size or for practical reasons, but also due to aesthetic reasons. While we have chosen the axioms A, NA1, and NA2 to capture the properties of the operator '&', and the axioms O1, O2, and NO to capture the properties of the operator '|', because these axioms naturally fit into our proof strategy, a popular choice in Mathematical Logic courses is to replace NA1, NA2, and NO with the following respective three alternative (sound) axioms, which could be argued to be a more aesthetic choice as each of these alternative axioms contains only implication, and not also negation, in addition to the relevant operator ('&' or '|') whose properties are captured:

AE1: '((p&q)→q)'

AE2: '((p&q)→p)'

OE: '((p→r)→((q→r)→((p|q)→r)))'

propositions/axiomatic_systems.py
```
# Alternatives for NA1, NA2, NO without negation

#: Conjunction elimination (right)
AE1 = InferenceRule([], Formula.parse('((p&q)->q)'))
#: Conjunction elimination (left)
AE2 = InferenceRule([], Formula.parse('((p&q)->p)'))
#: Disjunction elimination
OE = InferenceRule([], Formula.parse('((p->r)->((q->r)->((p|q)->r)))'))

#: Hilbert axiomatic system for all operators, consisting of the rules in
#: `HILBERT_AXIOMATIC_SYSTEM`, as well as `A`, `AE1`, `AE2`, `O1`, `O2`, `OE`,
#: `T`, and `NF`.
HILBERT_AXIOMATIC_SYSTEM_FULL = \
    frozenset(HILBERT_AXIOMATIC_SYSTEM.union({A, AE1, AE2, O1, O2, OE, T, NF}))
```

Your final optional task in this chapter is to show that the axiomatic system resulting from this replacement, $\hat{\mathcal{H}} = \{\mathrm{MP}, \mathrm{I1}, \mathrm{D}, \mathrm{N}, \mathrm{A}, \mathrm{AE1}, \mathrm{AE2}, \mathrm{O1}, \mathrm{O2}, \mathrm{OE}, \mathrm{T}, \mathrm{NF}\}$, indeed still gives rise to a complete proof system (even if our formulas may contain also '&', '|', 'T', and 'F'). You will once again rely on the power of the Lemma Theorem, by which it suffices to only show that NA1, NA2, and NO are provable via $\hat{\mathcal{H}}$.

Optional Task 9. Prove each of the following inference rules via the respective axioms specified below. Each of these proofs should be returned by the respective function specified below, whose missing code you should implement.

a. Prove NA1 via MP, I0, I1, D, N, and AE1. The proof should be returned by `prove_NA1()`.

   Hint: Use AE1.

b. Prove NA2 via MP, I0, I1, D, N, and AE2. The proof should be returned by `prove_NA2()`.

   Hint: Use AE2.

c. Prove NO via MP, I0, I1, D, N, and OE. The proof should be returned by `prove_NO()`.

   Hint: Use OE; make use of I2 (more than once) and of the inference rule that you have proved in Part f of Optional Task 7.

propositions/some_proofs.py

```
def prove_NA1() -> Proof:
    """Proves `NA1` via `MP`, `I0`, `I1`, `D`, `N`, and `AE1`.

    Returns:
        A valid proof of `NA1` via the inference rules `MP`, `I0`, `I1`, `D`,
        and `AE1`.
    """
    # Optional Task 6.9a

def prove_NA2() -> Proof:
    """Proves `NA2` via `MP`, `I0`, `I1`, `D`, `N`, and `AE2`.

    Returns:
        A valid proof of `NA2` via the inference rules `MP`, `I0`, `I1`, `D`,
        and `AE2`.
    """
    # Optional Task 6.9b

def prove_NO() -> Proof:
    """Proves `NO` via `MP`, `I0`, `I1`, `D`, `N`, and `OE`.

    Returns:
        A valid proof of `NO` via the inference rules `MP`, `I0`, `I1`, `D`, and
        `OE`.
    """
    # Optional Task 6.9c
```