Chapter 4:

Proof by Deduction

In this chapter, we will define the syntax of a **deductive proof**, i.e., a formal proof that starts from a set of assumptions, and proceeds step by step by inferring additional intermediate results, until the intended conclusion is inferred. Specifically, a significant portion of this chapter will be focused on verifying the validity of a given proof.

1 Inference Rules

Before getting into proofs, we will define the notion of an **inference rule** that allows us to proceed in a proof by **deducing** a "conclusion line" from previous "assumption lines". Moreover, what a proof proves is a "lemma" or a "theorem", which may itself be viewed as an inference rule, stating that the conclusion of the lemma or theorem follows from its assumptions.

Definition (Inference Rule)**.** An **inference rule** is composed of a list of zero or more propositional formulas called the **assumptions** of the rule, and one additional propositional formula called the **conclusion** of the rule.

An example of an inference rule is as follows: Assumptions: '(p|q)', '(~p|r)'; Conclusion: '(q|r)'. An inference rule need not necessarily have any assumptions. An example of an inference rule without assumptions (i.e., with zero assumptions) is as follows: (Assumptions: none;) Conclusion: '(~p|p)'.

The file `propositions/proofs.py` defines (among other classes) a Python class `InferenceRule` for holding an inference rule as a list of assumptions and a conclusion, all of type `Formula`.

propositions/proofs.py

```
@frozen
class InferenceRule:
    """An immutable inference rule in Propositional Logic, comprised of zero
    or more assumed propositional formulas, and a conclusion propositional
    formula.

    Attributes:
        assumptions: the assumptions of the rule.
        conclusion: the conclusion of the rule.
    """
    assumptions: Tuple[Formula, ...]
    conclusion: Formula

    def __init__(self, assumptions: Sequence[Formula], conclusion: Formula):
        """Initializes an `InferenceRule` from its assumptions and conclusion.

        Parameters:
```

```
            assumptions: the assumptions for the rule.
            conclusion: the conclusion for the rule.
        """
        self.assumptions = tuple(assumptions)
        self.conclusion = conclusion
```

Task 1. Implement the missing code for the method `variables()` (of class `InferenceRule`), which returns all of the variables that appear in any of the assumptions and/or in the conclusion of the rule.

propositions/proofs.py

```
class InferenceRule:
            ⋮
    def variables(self) -> Set[str]:
        """Finds all atomic propositions (variables) in the current inference
        rule.

        Returns:
            A set of all atomic propositions used in the assumptions and in the
            conclusion of the current inference rule.
        """
        # Task 4.1
```

Examples: Taking `rule` to be the first inference rule (the one with two assumptions) given as an example above, `rule.variables()` should return `{'p', 'q', 'r'}`, and for the second inference rule (the assumptionless one) given as an example above it should return `{'p'}`.

In this chapter we will allow for arbitrary inference rules (arbitrary assumptions and arbitrary conclusion) and focus solely on the *syntax* of using them in deductive proofs. This syntax in particular will not depend much on whether any of these inference rule is semantically "correct" or not. In later chapters, we will however be more specific about our inference rules, and the first requirement that we will want is for all of them to indeed be semantically **sound**:

Definition (Soundness)**.** We say that a set of assumptions A **entails** a conclusion ϕ if every model that satisfies all the assumptions in A also satisfies ϕ. We denote this by $A \models \phi$.[1] We say that the inference rule whose assumptions are the elements of the set A and whose conclusion is ϕ is **sound** if $A \models \phi$.

For example, it is easy to verify that {'p','(p→q)'} $\models$ 'q', and thus the inference rule with assumptions 'p' and '(p→q)' and conclusion 'q' is sound.[2] Similarly, the two inference rules given as examples above are also sound. On the other hand, the inference rule with the single assumption '(p→q)' and the conclusion '(q→p)' is not sound since the model that assigns *False* to p and *True* to q satisfies the assumption but not the conclusion. If A is a singleton set, then we sometimes remove the set notation and write, for example, '~~p' $\models$ 'p' (called **Double-Negation Elimination**). If A is the empty

[1] The symbol $\models$ is sometimes used also in a slightly different way: for a model M and a formula ϕ one may write $M \models \phi$ (i.e., M is a model of ϕ) to mean that ϕ evaluates to *True* in the model M. For example, {'p':*True*,'q':*False*} $\models$ '(p&~q)'.

[2] This inference rule is called **Modus Ponens**, and will be of major interest starting in the next chapter.

set then we simply write $\models \phi$, which is equivalent to saying that ϕ is a tautology. Thus, for example, $\models$ '(p|~p)' (called the **Law of Excluded Middle**).

The next two tasks explore the semantics of inference rules. Accordingly, the functions that you are asked to implement in these tasks should be implemented in the file `propositions/semantics.py`.

Task 2. Implement the missing code for the function `evaluate_inference(rule, model)`, which returns whether the given inference rule holds in the given model, that is, whether it is *not* the case that all assumptions hold in this model but the conclusion does not.

propositions/semantics.py

```
def evaluate_inference(rule: InferenceRule, model: Model) -> bool:
    """Checks if the given inference rule holds in the given model.

    Parameters:
        rule: inference rule to check.
        model: model to check in.

    Returns:
        ``True`` if the given inference rule holds in the given model, ``False``
        otherwise.

    Examples:
        >>> evaluate_inference(InferenceRule([Formula('p')], Formula('q')),
        ...                    {'p': True, 'q': False})
        False

        >>> evaluate_inference(InferenceRule([Formula('p')], Formula('q')),
        ...                    {'p': False, 'q': False})
        True
    """
    assert is_model(model)
    # Task 4.2
```

Task 3. Implement the missing code for the function `is_sound_inference(rule)`, which returns whether the given inference rule is sound, i.e., whether it holds in every model.

propositions/semantics.py

```
def is_sound_inference(rule: InferenceRule) -> bool:
    """Checks if the given inference rule is sound, i.e., whether its
    conclusion is a semantically correct implication of its assumptions.

    Parameters:
        rule: inference rule to check.

    Returns:
        ``True`` if the given inference rule is sound, ``False`` otherwise.
    """
    # Task 4.3
```

2 Specializations of an Inference Rule

We will usually think of an inference rule as a template where the variables serve as placeholders for formulas. For example if we look at the Double-Negation Elimination rule, ‘~~p’ $\models$ ‘p’, we may plug any formula into the variable ‘p’ and get a “special case” or a **specialization** of the rule, so for example we may substitute ‘(q→r)’ for ‘p’ and get the following inference rule: ‘~~(q→r)’ $\models$ ‘(q→r)’, or we may substitute ‘x’ for ‘p’ and get the inference rule ‘~~x’ $\models$ ‘x’, both of which are specializations of the original inference rule.

Definition (Specialization)**.** An inference rule β is a **specialization** of an inference rule α if there exist a number of formulas $\phi_1, \ldots, \phi_n$ and a matching number of variables $v_1, \ldots, v_n$, such that β is obtained from α by (simultaneously) substituting the formula ϕ_i for each occurrence of the variable v_i in all of the assumptions of α (while maintaining the order of the assumptions) as well as in its conclusion.

Given an inference rule and a desired substitution/specialization map, it is quite easy to obtain the specialized inference rule.

Task 4. Implement the missing code for the method `specialize(specialization_map)` (of class `InferenceRule`), which returns the specialization of the inference rule according to the given specialization map.

propositions/proofs.py

```
#: A mapping from variable names to formulas.
SpecializationMap = Mapping[str, Formula]

⋮
class InferenceRule:
            ⋮
    def specialize(self, specialization_map: SpecializationMap) -> \
            InferenceRule:
        """Specializes the current inference rule by simultaneously substituting
        each variable `v` that is a key in `specialization_map` with the
        formula `specialization_map[v]`.

        Parameters:
            specialization_map: mapping defining the specialization to be
                performed.

        Returns:
            The resulting inference rule.
        """
        for variable in specialization_map:
            assert is_variable(variable)
        # Task 4.4
```

Hint: The `substitute_variables()` method of class `Formula` performs a similar substitution for a single formula.

Given two inference rules, it is only slightly more difficult to tell whether one is a specialization of the other. First, the number of assumptions should match. Then, for every formula in the assumptions or the conclusion there should be a match between the formula in the “general” rule and the corresponding formula in the alleged specialization: if the “general” formula is a variable then the specialized formula may be any formula. Otherwise, the root of the specialized formula must be identical to the root of the general formula, and the subtrees should match recursively according to the same conditions.

Moreover, there is an important additional consistency condition: all occurrences of each variable in the general rule must correspond to the same subformula throughout the specialization. The following task implements this procedure.

Task 5. In this task you will not only determine whether a given inference rule is a specialization of another, but you will also, if this is the case, find the appropriate specialization map. We will once again represent a specialization map as a Python dictionary (mapping variables of the general rule to sub-formulas of the specialized rule), and use Python's `None` value to represent that no specialization map exists since the alleged specialization is in fact not a specialization of a given general rule.

a. Start with the basic check that ensures all occurrences of each variable are consistently mapped to the same subformula. Implement the missing code for the function `_merge_specialization_maps(specialization_map1, specialization_map2)` (a static method of class `InferenceRule`), which takes two specialization maps, checks whether they are consistent with each other in the sense that no variable appears in both but is mapped to a different formula in each, and if so, returns the merger of the maps, and otherwise returns `None`.

propositions/proofs.py

```
class InferenceRule:
            ⋮
    @staticmethod
    def _merge_specialization_maps(
            specialization_map1: Union[SpecializationMap, None],
            specialization_map2: Union[SpecializationMap, None]) -> \
            Union[SpecializationMap, None]:
        """Merges the given specialization maps while checking their
        consistency.

        Parameters:
            specialization_map1: first mapping to merge, or ``None``.
            specialization_map2: second mapping to merge, or ``None``.

        Returns:
            A single mapping containing all (key, value) pairs that appear in
            either of the given maps, or ``None`` if one of the given maps is
            ``None`` or if some key appears in both given maps but with
            different values.
        """
        if specialization_map1 is not None:
            for variable in specialization_map1:
                assert is_variable(variable)
        if specialization_map2 is not None:
            for variable in specialization_map2:
                assert is_variable(variable)
        # Task 4.5a
```

b. Next, complete the missing code for the function that figures out which specialization map (if any) makes a given formula a specialization of another. This is captured by the function `_formula_specialization_map(general, specialization)` (a static method of class `InferenceRule`), which takes two formulas and returns such a map if the second given formula is indeed a specialization of the first, and `None` otherwise.

propositions/proofs.py
```
class InferenceRule:
          ⋮
    @staticmethod
    def _formula_specialization_map(general: Formula, specialization: Formula) \
            -> Union[SpecializationMap, None]:
        """Computes the minimal specialization map by which the given formula
        specializes to the given specialization.

        Parameters:
            general: non-specialized formula for which to compute the map.
            specialization: specialization for which to compute the map.

        Returns:
            The computed specialization map, or ``None`` if `specialization` is
            in fact not a specialization of `general`.
        """
        # Task 4.5b
```

Hint: Use the function `_merge_specialization_maps()` that you have just written.

c. Finally, complete the missing code for the general method that tells if and how one inference rule is a specialization of another, captured by the method `specialization_map(specialization)` of class `InferenceRule`, which takes an alleged specialization of the rule, and returns the corresponding specialization map, or `None` if the alleged specialization is in fact not a specialization of the current rule. Remember that the definition of a specialization requires that the order of the assumptions be preserved.

propositions/proofs.py
```
class InferenceRule:
          ⋮
    def specialization_map(self, specialization: InferenceRule) -> \
            Union[SpecializationMap, None]:
        """Computes the minimal specialization map by which the current
        inference rule specializes to the given specialization.

        Parameters:
            specialization: specialization for which to compute the map.

        Returns:
            The computed specialization map, or ``None`` if `specialization` is
            in fact not a specialization of the current rule.
        """
        # Task 4.5c
```

Note that if we just want to tell whether one rule is or is not a specialization of another, we just need to check whether `specialization_map()` returns a specialization map rather than `None`, which we have already implemented for you as a method of class `InferenceRule`.

propositions/proofs.py
```
class InferenceRule:
          ⋮
```

```
    def is_specialization_of(self, general: InferenceRule) -> bool:
        """Checks if the current inference rule is a specialization of the given
        inference rule.

        Parameters:
            general: non-specialized inference rule to check.

        Returns:
            ``True`` if the current inference rule is a specialization of
            `general`, ``False`` otherwise.
        """
        return general.specialization_map(self) is not None
```

3 Deductive Proofs

We are now ready to introduce the main concept of this chapter, the (deductive) proof. Such a proof should be a syntactic derivation of one inference rule (the "lemma" being proven) using a set of other inference rules (which we already take as given). We will use the very standard form of a proof that proceeds line by line. Each line in the proof may either be a direct quote of one of the assumptions of the lemma that we are proving, or may be derived from previous lines in the proof using a specialization of one of the inference rules that the proof may use. The last line of the proof should exactly be the conclusion of the lemma that we are proving. As noted above, in this chapter we will allow for arbitrary inference rules to be specified for use by a proof, and so we will explicitly specify the set of inference rules that may be used in each proof.[3] Here is an example of a proof:

Lemma to be proven: Assumption: '(x|y)'; Conclusion: '(y|x)'.

Inference rules allowed:

1. Assumptions: '(p|q)', '(~p|r)'; Conclusion: '(q|r)'.
2. (Assumptions: none;) Conclusion: '(~p|p)'.

Proof:

1. '(x|y)' (Assumption of the lemma to be proven)
2. '(~x|x)' (Specialization of Inference Rule 2)
3. '(y|x)' (Specialization of Inference Rule 1; Assumption 1: Line 1, Assumption 2: Line 2.)

When we quote an assumption of the lemma to be proven, we must quote it verbatim, with no substitutions whatsoever. Thus, for example, Line 1 of the above proof precisely quotes the assumption '(x|y)', and could *not* have quoted instead, say, '(w|z)', which indeed does *not* follow from the assumption '(x|y)'. On the other hand, when we use an inference rule to derive a line from previous lines, our derivation can use any specialization of that rule. Thus, for example, Line 2 of the above proof uses the inference rule with no

[3] As we progress in the following chapters, we will converge to a single specific set of rules that will be used from then onward.

assumptions and conclusion '(~p|p)' to derive '(~x|x)' from an empty set of assumptions, as this is a specialization obtained from that inference rule by substituting the formula 'x' for the variable 'p'. Similarly, Line 3 of the above proof uses an inference rule with assumptions '(x|y)' and '(~x|x)' and conclusion '(y|x)', which is a specialization of the first allowed inference rule, obtained by substituting 'x' for 'p' and for 'r', and 'y' for 'q'.

Definition ($\vdash_{\mathcal{R}}$)**.** Given a set of inference rules $\mathcal{R}$, we say that the inference rule whose assumptions are the elements of the set A and whose conclusion is ϕ is **provable** via $\mathcal{R}$ if there exists a (valid) proof of that rule using $\mathcal{R}$ as the set of allowed inference rules. We denote this by $A \vdash_{\mathcal{R}} \phi$.

We emphasize that the notion of a proof is completely syntactic, and therefore so is the definition of $A \vdash_{\mathcal{R}} \phi$. However, when we think about a "proof" we intuitively desire also some semantic property: that a "proof" indeed "proves" what it claims. That is, that if we have a "proof" of some rule from correct assumptions, then indeed the conclusion of the "proof" is also correct; in other words, that the rule that was proven is sound. As we will see below, the notion of proof that we just described indeed has this property, as long as it is *only allowed to use sound inference rules.* But for now, let us proceed to handle the syntax.

The file `propositions/proofs.py` also defines a class `Proof` for holding a deductive proof. Each lines of the proof, including a full justification, is held by the class `Proof.Line` define in the same file. (Note that unlike in the proof example given above, in the code all line indices are 0-based.)

propositions/proofs.py

```
@frozen
class Proof:
    """An immutable deductive proof in Propositional Logic, comprised of a
    statement in the form of an inference rule, a set of inference rules that
    may be used in the proof, and a list of lines that prove the statement via
    these inference rules.

    Attributes:
        statement: the statement of the proof.
        rules: the allowed rules of the proof.
        lines: the lines of the proof.
    """
    statement: InferenceRule
    rules: FrozenSet[InferenceRule]
    lines: Tuple[Proof.Line, ...]

    def __init__(self, statement: InferenceRule,
                 rules: AbstractSet[InferenceRule],
                 lines: Sequence[Proof.Line]):
        """Initializes a `Proof` from its statement, allowed inference rules,
        and lines.

        Parameters:
            statement: the statement for the proof.
            rules: the allowed rules for the proof.
            lines: the lines for the proof.
        """
        self.statement = statement
        self.rules = frozenset(rules)
        self.lines = tuple(lines)
```

```
@frozen
class Line:
    """An immutable line in a deductive proof, comprised of a formula that
    is either justified as an assumption of the proof, or as the conclusion
    of a specialization of an allowed inference rule of the proof, the
    assumptions of which are justified by previous lines in the proof.

    Attributes:
        formula: the formula justified by the line.
        rule: the inference rule, out of those allowed in the proof, that
            has a specialization that concludes the formula; or ``None`` if
            the formula is justified as an assumption of the proof.
        assumptions: tuple of zero or more numbers of previous lines in the
            proof whose formulas are the respective assumptions of the
            specialization of the rule that concludes the formula, if the
            formula is not justified as an assumption of the proof.
    """
    formula: Formula
    rule: Optional[InferenceRule]
    assumptions: Optional[Tuple[int, ...]]

    def __init__(self, formula: Formula,
                 rule: Optional[InferenceRule] = None,
                 assumptions: Optional[Sequence[int]] = None):
        """Initializes a `Proof.Line` from its formula, and optionally its
        rule and numbers of justifying previous lines.

        Parameters:
            formula: the formula to be justified by the line.
            rule: the inference rule, out of those allowed in the proof,
                that has a specialization that concludes the formula; or
                ``None`` if the formula is to be justified as an assumption
                of the proof.
            assumptions: numbers of previous lines in the proof whose
                formulas are the respective assumptions of the
                specialization of the rule that concludes the formula, or
                ``None`` if the formula is to be justified as an assumption
                of the proof.
        """
        assert (rule is None and assumptions is None) or \
               (rule is not None and assumptions is not None)
        self.formula = formula
        self.rule = rule
        if assumptions is not None:
            self.assumptions = tuple(assumptions)

    def is_assumption(self) -> bool:
        """Checks if the current proof line is justified as an assumption of
        the proof.

        Returns:
            ``True`` if the current proof line is justified as an assumption
            of the proof, ``False`` otherwise.
        """
        return self.rule is None
```

Task 6. The goal of this task is to check whether a given (alleged) proof is indeed a valid one.

a. Start by implementing the missing code for the method `rule_for_line(line_number)` (of class `Proof`), which returns an inference rule comprised of the specified line and all the lines by which it is justified.

propositions/proofs.py
```
class Proof:
        ⋮
    def rule_for_line(self, line_number: int) -> Union[InferenceRule, None]:
        """Computes the inference rule whose conclusion is the formula justified
        by the specified line, and whose assumptions are the formulas justified
        by the lines specified as the assumptions of that line.

        Parameters:
            line_number: number of the line according to which to compute the
                inference rule.

        Returns:
            The computed inference rule, with assumptions ordered in the order
            of their numbers in the specified line, or ``None`` if the specified
            line is justified as an assumption.
        """
        assert line_number < len(self.lines)
        # Task 4.6a
```

b. Continue by implementing the missing code for the method `is_line_valid(line_number)` (of class `Proof`), which returns whether the specified line is either an assumption and justified as such, or is indeed the result of applying a specialization of the inference rule by which the line is justified to the previous lines by which the line is justified.

propositions/proofs.py
```
class Proof:
        ⋮
    def is_line_valid(self, line_number: int) -> bool:
        """Checks if the specified line validly follows from its justifications.

        Parameters:
            line_number: number of the line to check.

        Returns:
            If the specified line is justified as an assumption, then ``True``
            if the formula justified by this line is an assumption of the
            current proof, ``False`` otherwise. Otherwise (i.e., if the
            specified line is justified as a conclusion of an inference rule),
            ``True`` if the rule specified for that line is one of the allowed
            inference rules in the current proof, and it has a specialization
            that satisfies all of the following:

            1. The conclusion of that specialization is the formula justified by
               that line.
            2. The assumptions of this specialization are the formulas justified
               by the lines that are specified as the assumptions of that line
               (in the order of their numbers in that line), all of which must
               be previous lines.
```

```
        """
        assert line_number < len(self.lines)
        # Task 4.6b
```

c. Finally, implement the missing code for the method `'is_valid()'` (of class `Proof`), which returns whether the proof is a valid proof of the objective inference rule (the "lemma" to be proven), using the allowed inference rules.

propositions/proofs.py

```
class Proof:
        ⋮
    def is_valid(self) -> bool:
        """Checks if the current proof is a valid proof of its claimed statement
        via its inference rules.

        Returns:
            ``True`` if the current proof is a valid proof of its claimed
            statement via its inference rules, ``False`` otherwise.
        """
        # Task 4.6c
```

4 Practice Proving

Before continuing with our agenda of reasoning about the formal deductive proofs that we have just defined, it is worthwhile to first get comfortable in simply using them. Here are two basic exercises in writing formal proofs using the `Proof` class. The solutions should be implemented in the file `propositions/some_proofs.py`. We warmly recommend to first try and figure out the proof strategy with a pen and a piece of paper, and only then write the code that returns the appropriate `Proof` object.

Task 7. Prove the following inference rule: Assumption: '(p&q)'; Conclusion: '(q&p)'; via the following three inference rules:

- Assumptions: 'x', 'y'; Conclusion: '(x&y)'.
- Assumptions: '(x&y)'; Conclusion: 'x'.
- Assumptions: '(x&y)'; Conclusion: 'y'.

The proof should be returned by the function `prove_and_commutativity()` (in the file `propositions/some_proofs.py`), whose missing code you should implement.

propositions/some_proofs.py

```
# Some inference rules that only use conjunction.

#: Conjunction introduction inference rule
A_RULE = InferenceRule([Formula.parse('x'), Formula.parse('y')],
                       Formula.parse('(x&y)'))
#: Conjunction elimination (right) inference rule
AE1_RULE = InferenceRule([Formula.parse('(x&y)')],Formula.parse('y'))
#: Conjunction elimination (left) inference rule
AE2_RULE = InferenceRule([Formula.parse('(x&y)')],Formula.parse('x'))

def prove_and_commutativity() -> Proof:
```

```
    """Proves '(q&p)' from '(p&q)' via `A_RULE`, `AE2_RULE`, and `AE1_RULE`.

    Returns:
        A valid proof of '(q&p)' from the single assumption '(p&q)' via the
        inference rules `A_RULE`, `AE2_RULE`, and `AE1_RULE`.
    """
    # Task 4.7
```

The next and final task requires some more ingenuity. It focuses on inference rules that only involve the *implies* operator, and uses the following three inference rules, which in the following chapters will end up being part of our "chosen" set of inference rules (which, as we will see in Chapter 6, suffice for proving all sound inference rules).

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

I1: (Assumptions: none;) Conclusion: '(q→(p→q))'

D: (Assumptions: none;) Conclusion: '((p→(q→r))→((p→q)→(p→r)))'

These inference rules, alongside the rule that you are asked to prove in the next task, are defined in the file `propositions/axiomatic_systems.py`.[4]

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)))'))
```

Task 8. Prove the following inference rule via the inference rules MP, I1, and D:

I0: (Assumptions: none;) Conclusion: '(p→p)'

The proof should be returned by the function `prove_I0()` (in the file `propositions/some_proofs.py`), whose missing code you should implement.

propositions/some_proofs.py

```
def prove_I0() -> Proof:
    """Proves `I0` via `MP`, `I1`, and `D`.

    Returns:
        A valid proof of `I0` via the inference rules `MP`, `I1`, and `D`.
    """
    # Task 4.8
```

Hint: Start by using the rule D with '(p→p)' substituted for 'q' and with 'p' substituted for 'r'. Notice that this would give you something that looks like '(ϕ→(ψ→(p→p)))'. Now try to extract the required '(p→p)' using the rules MP and I1.

[4]You will not be asked to implement anything in this file throughout this course.

5 The Soundness Theorem

Let us emphasize again what should be clear up to this point: the validity of a proof is a purely syntactic matter. However at this point we are ready for the first relation between the syntactic world of proofs and the semantic world of truths: anything that is provable via sound inference rules must be true, i.e., must be sound.

Theorem (The Soundness Theorem for Propositional Logic)**.** *Any inference rule that is provable via (only) sound inference rules is itself sound as well. That is, if $\mathcal{R}$ contains only sound inference rules, and if $A \vdash_{\mathcal{R}} \phi$, then $A \models \phi$.*

This trivial-yet-magical theorem provides in fact the basic justification for the whole concept of Mathematics: proving something as a way of knowing that it is true. Otherwise, there would have been no point in proving anything! This theorem also indicates why we will always only allow using sound inference rules in our proofs, as otherwise we might prove claims that are wrong. The following two tasks prove the Soundness Theorem. The functions that you are asked to implement in these tasks should be implemented in the file `propositions/soundness.py`.

Our first order of business is to tackle the use of specializations in a proof: recall that a proof that uses a set $\mathcal{R}$ of inference rules is allowed to use, in every line, any specialization of any rule in $\mathcal{R}$ rather than just these rules verbatim. This turns out not to be an issue, since if we start with any sound inference rule like 'x $\models$ ~~x' and then "plug into x" any formula, for example '(p&q)', then we get a *sound* specialization: '(p&q) $\models$ ~~(p&q)'. The reason for this is that had there been a counterexample to the specialized inference rule, then it would directly yield a counterexample to the original inference rule as well. The following task makes this explicit.

Task 9. Implement the missing code for the function `rule_nonsoundness_from_specialization_nonsoundness(general, specialization, model)`, which takes an inference rule, a specialization of this rule, and a model that is a counterexample to the soundness of this specialization, and returns a model that is a counterexample to the soundness of the general inference rule.

propositions/soundness.py

```
def rule_nonsoundness_from_specialization_nonsoundness(
        general: InferenceRule, specialization: InferenceRule, model: Model) \
        -> Model:
    """Demonstrated the non-soundness of the given general inference rule given
    an example of the non-soundness of the given specialization of this rule.

    Parameters:
        general: inference rule to the soundness of which to find a
            counterexample.
        specialization: non-sound specialization of `general`.
        model: model in which `specialization` does not hold.

    Returns:
        A model in which `general` does not hold.
    """
    assert specialization.is_specialization_of(general)
    assert not evaluate_inference(specialization, model)
    # Task 4.9
```

Hint: This function will be tested on inference rules with many variables, so iterating over all models to find a model to return (similarly to your implementation of `is_sound_inference()`) is not an adequate solution strategy (and more importantly, does not programmatically prove what we have set out to prove). Instead, try to understand how to use the given model to find a suitable model to return.

Your solution to Task 9 proves the following lemma.

Lemma (Specialization Soundness)**.** *Every specialization of a sound inference rule is itself sound as well.*

Once we have this lemma under our belt we can proceed to prove the Soundness Theorem. Assume by way of contradiction that we have a proof that starts with a set of assumptions and proves a conclusion that is not semantically entailed by them. If we look at any model that purports to be a counterexample to this proved inference rule then we can obtain from it a counterexample to one of the inference rules that are used in the proof. Which one? Look at the sequence of lines of the proof. In the beginning of the proof we have assumptions that evaluate to *True* in the model and at the end of the proof we have a conclusion that evaluates to *False* in this model (this is exactly what it means for the model to be a counterexample to the proved inference rule). If we look at the first line in the proof that evaluates to *False* in the model, it must have used an inference rule that is not sound, since it if were sound, then a model that satisfies all of its assumptions (like our counterexample model) would have also satisfied its conclusion. The following task makes this explicit.

Task 10 (Programmatic Proof of the Soundness Theorem)**.** Implement the missing code for the function `nonsound_rule_of_nonsound_proof(proof, model)`, which takes a valid proof and a model that is a counterexample to the statement of the given proof, and returns a non-sound inference rule that is used in the given proof, along with a model that is a counterexample to the soundness of the returned inference rule.

propositions/soundness.py

```
def nonsound_rule_of_nonsound_proof(proof: Proof, model: Model) -> \
        Tuple[InferenceRule, Model]:
    """Finds a non-sound inference rule used by the given valid proof of a
    non-sound inference rule, and demonstrates the non-soundness of the former
    rule.

    Parameters:
        proof: valid proof of a non-sound inference rule.
        model: model in which the inference rule proved by the given proof does
            not hold.

    Returns:
        A pair of a non-sound inference rule used in the given proof and a model
        in which this rule does not hold.
    """
    assert proof.is_valid()
    assert not evaluate_inference(proof.statement, model)
    # Task 4.10
```

Hint: This function will be tested on proofs with inference rules with many variables, so running `is_sound_inference()` on each inference rule used in the given proof is not an adequate solution strategy (and more importantly, does not programmatically prove the Soundness Theorem). Instead, try to understand how to use the given model to find a suitable rule (and model) to return.

The Soundness Theorem gives us a clear one-sided connection between the syntactic notion of $A \vdash_{\mathcal{R}} \phi$ and the semantic notion of $A \models \phi$, i.e., that the former implies the latter. Our goal in the next two chapters will be to prove a converse called the **Completeness Theorem**: that $A \models \phi$ implies $A \vdash_{\mathcal{R}} \phi$ for some specific small set of sound inference rules $\mathcal{R}$.