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 by 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: Iterable[Formula], conclusion: Formula) -> \
        None:
        """Initialized 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. 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 set then we

[1] We sometimes use the symbol $\models$ also in a slightly different way: for a model M and a formula ϕ we write $M \models \phi$ (i.e., M is a model of ϕ) if ϕ evaluates to *True* in the model M. Thus, 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.

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 you are asked to implement in these tasks are 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 i is a **specialization** of an inference rule j 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 i is obtained from j by (simultaneously) substituting the formula ϕ_1 for each occurrence of the variable v_1, the formula ϕ_2 for each occurrence of the variable v_2, etc., in all of the assumptions of j (while maintaining the order of the assumptions) as well as in its conclusion.

What should be quite clear is that specializing an inference rule preserves the property of being sound.

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

Proof. Let us start with a sound inference rule $A \models \phi$ with variables $v_1, \ldots, v_n$, and let us consider its specialization obtained by substituting, for every i, the formula ϕ_i for the variable v_i in all assumptions and in the conclusion. Consider a model M over the set of variables of the specialization (i.e., the set of variables of all the ϕ_is)[3] that satisfies all of the assumptions of the specialization. We need to show that M also satisfies the conclusion of the specialization. For every i, denote the truth value of ϕ_i in M by b_i. The truth value in M of each assumption of the specialization is, by definition, given by induction over the formula, so it may be equivalently calculated by first evaluating each ϕ_i (to get the value b_i) and then evaluating the original assumption (on the v_is) in the model M' that assigns the value b_i to each v_i. The truth value in M of the conclusion of the specialization is similarly defined by induction over the formula, so it may also be equivalently calculated by first evaluating each ϕ_i (to get the value b_i) and then evaluating the original conclusion (on the v_is) in the same model M' that assigns the value b_i to each v_i. So, since all assumptions of the specialization evaluate to *True* in M, all of the assumptions of the original inference rule evaluate to *True* in M'. Therefore, since the original inference rule is sound, also the conclusion of the original inference rule evaluates to *True* in M'. Therefore, the conclusion of the specialization evaluates to *True* in M. □

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:
```

[3]We can assume without loss of generality that we substitute each of the original variables v_i with some formula ϕ_i, because we can always take ϕ_i for some specific i to be the same as v_i, substituting a variable for itself, which has the same effect as not substituting that variable.

```
          ⋮
    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
```

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 subformulas 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 map to merge, or ``None``.
            specialization_map2: second map to merge, or ``None``.

        Returns:
            A single map 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. Hint: use the function `merge_specialization_maps()` that you have just written.

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
```

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.[4] Here is an

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

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 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 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. It is very easy to see that the notion of proof that we just described has this property, as long as it is only allowed to use sound inference rules.

Theorem (The Soundness Theorem for Propositional Logic)**.** *Any inference rule that can be proven 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 $A \models \phi$.*

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 $A \models \phi$. 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 $\mathcal{R}$. For now, here is the immediate proof of the Soundness Theorem, which is the easier of these two converse theorems.

Proof. Since $A \vdash_{\mathcal{R}} \phi$, there exists a proof of ϕ from A via $\mathcal{R}$. In order to show that $A \models \phi$, we need to show that for every model (on the variables of all formulas in A and of ϕ) in which all of the assumption in A evaluate to *True*, the conclusion ϕ also evaluates to *True*. So fix such a model. We will actually show that all the formulas in all the lines of the proof of ϕ evaluate to *True*, and since, by definition, the conclusion ϕ must be the last line of the proof, then ϕ in particular evaluates to *True*, as claimed.

To prove that all lines in the proof evaluate to *True*, we will proceed line by line (formally, using induction on the line number). Look at some line number i. If Line i is an assumption, then since all assumptions evaluate to *True* in the fixed model, then so does Line i. Otherwise, i is derived from previous lines using some specialization of an inference rule $R \in \mathcal{R}$. Since R is sound, then by the Specialization Lemma so is its specialization, so since all of the assumptions of the specializations are previous lines and thus, we already know, evaluate to *True*, the conclusion of the specialization must evaluate to *True* as well, as we needed to show. $\square$

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, 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 proof in the form of 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.
    """
    statment: InferenceRule
    rules: FrozenSet[InferenceRule]
    lines: Tuple[Proof.Line, ...]

    def __init__(self, statement: InferenceRule,
                 rules: AbstractSet[InferenceRule],
                 lines: Iterable[Proof.Line]) -> None:
        """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, a
                specialization of which concludes the formula, or ``None`` if
                the formula is justified as an assumption of the proof.
            assumptions: a tuple of zero or more indices 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[Iterable[int]] = None) -> None:
            """Initializes a `Proof.Line` from its formula, and optionally its
            rule and indices of justifying previous lines.

            Parameters:
                formula: the formula to be justified by this line.
                rule: the inference rule out of those allowed in the proof, a
                    specialization of which concludes the formula, or ``None``
                    if the formula is to be justified as an assumption of the
                    proof.
                assumptions: an iterable over indices 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: index of the line according to which to construct the
                inference rule.

        Returns:
            The constructed inference rule, with assumptions ordered in the
            order of their indices 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: index 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),
            then ``True`` if and only if all of the following hold:

            1. The rule specified for that line is one of the allowed inference
               rules in the current proof.
            2. Some specialization of the rule specified for that line has
               the formula justified by that line as its conclusion, and the
               formulas justified by the lines specified as the assumptions of
               that line (in the order of their indices in this line) as its
               assumptions.
        """
        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

We conclude this chapter with 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', '$(p\rightarrow q)$'; Conclusion: 'q'

I1: (Assumptions: none;) Conclusion: '$(q\rightarrow(p\rightarrow q))$'

D: (Assumptions: none;) Conclusion: '$((p\rightarrow(q\rightarrow r))\rightarrow((p\rightarrow q)\rightarrow(p\rightarrow r)))$'

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`.[5]

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 fules MP, I1, and D:

I0: (Assumptions: none;) Conclusion: '$(p\rightarrow p)$'

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 '$(p\rightarrow p)$' substituted for 'q' and 'p' substituted for 'r'. Notice that this would give you something that looks like '$(\phi\rightarrow(\psi\rightarrow(p\rightarrow p)))$'. Now try to extract the required '$(p\rightarrow p)$' using the rules MP and I1.

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