Bachelor thesis
Computing Science

Radboud University

Formalizing and proving knights and knaves puzzles in three valued logic in Coq

Author:
Kimberley Frings
s1027103

First supervisor/assessor:
Dr. Perry Groot
`perry.groot@science.ru.nl`

Second assessor:
Dr. Engelbert Hubbers
`e.hubbers@cs.ru.nl`

January 17, 2022

Abstract

In this research, we introduce a methodology of formalizing and proving knights and knaves puzzles in Coq. We also show a formalization of a variation on classical knights and knaves puzzles, which introduces 'normal' people. In order to prove the solutions to these variations, we implement a three valued logic system and accompanying tactics in Coq.

Contents

Chapter 1

Introduction

Logical puzzles have been around for centuries. Be it for fun, or to make sense of the world, different types of logical puzzles have found different uses. One of the more popular logical puzzles is the knights and knaves puzzle. In these puzzles, a distinction is made between two types of people. One type, the knights, always speak the truth. The other type, the knaves, always lie. The puzzles generally contain a claim made by one or more people, of which we do not know whether they are a knight or a knave. The goal of the puzzle is to figure out, given the claims and by means of logical reasoning, what the type of each person is, if this is possible.

An extension on knights and knaves puzzles can be created by introducing a third type of person: a normal. A 'normal' person sometimes lies, and sometimes speaks the truth, but you can never know which it is. The logical puzzle becomes more complex, because the puzzler now has to take the uncertainty of these normal people into account. Nevertheless, a variety of knights and knaves puzzles exist with the inclusion of this third type of person, and each one of them can be solved using logical reasoning.

In the past, computers have shown to be a great help in the solving of logical problems. Some logical problems can even be solved automatically by using automatic deduction systems or satisfiability solvers. Given this providence in logical reasoning, it makes sense to incorporate computers in solving these types of puzzles. Not necessarily for a person who is trying to solve the puzzle, as this would spoil all the fun. However, from a puzzle maker's perspective it could be very useful to be able to prove, with the help of computers, the correctness of a puzzle.

In this thesis, we introduce an implementation of classical knights and knaves puzzles in the Coq proof assistant. We show its ability to solve these problems straightforward and out-of-the-box. Moreover, we give an implementation of a three valued logic system in Coq, where we introduce an ‘unknown’ value alongside the standard ‘true’ and ‘false’. With this three-valued logic system, we also introduce a set of deductive rules especially made to reason with three valued formulas. We use this three valued logic system to implement the variation on classical knights and knaves puzzles, where we assign the truth value of ‘unknown’ to a ‘normal’ person. We show that we can then use our custom set of deductive rules to solve knights and knaves puzzles of this variation.

In Chapter 2 we will start by introducing preliminary knowledge on different aspects of this thesis. Then, in Chapter 3 will will see a detailed description of the research we have carried out. In Chapter 4 we will compare our research to other existing literature. In Chapter 5 we will conclude our research and finally, in Chapter 6 we will take a look at future research opportunities.

Chapter 2

Preliminaries

2.1 Classical two-valued logic

Classical logic consists of propositions, which can have the value true or false. Besides these two basic propositions of true and false, we can also combine propositions into compound propositions using logical operators. Logical operators can be defined in terms of truth tables, which indicate the result of the operator for every combination of inputs.

Usually in classical logic, three basic operators are defined: $\neg$ (negation), $\wedge$ (conjunction), and $\vee$ (disjunction). The truth tables for these operators can be found in Table 2.1.

Table 2.1: Truth tables for negation, conjunction, and disjunction in classical logic.

A	¬ A
T	F
F	T

A	B	A ∧ B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A ∨ B
T	T	T
T	F	T
F	T	T
F	F	F

Building from these basic propositions, we can also define the $\rightarrow$ (implication) and the $\leftrightarrow$ (bi-implication) operators:

$$A \rightarrow B \equiv \neg A \vee B \qquad A \leftrightarrow B \equiv A \rightarrow B \wedge B \rightarrow A$$

In our classical logic system, propositions and their values are bound by the three Laws of Thought [1]:

1. **The Law of Excluded Middle**
   For every proposition it holds that it is either true or false:
   $$\forall P : P \lor \neg P$$

2. **The Law of Identity**
   For every proposition it holds that it always retains to the same truth value:
   $$\forall P : P \rightarrow P$$

3. **The Law of Contradiction**
   For no proposition it holds that it is true or false at the same time:
   $$\forall P : \neg(P \land \neg P)$$

2.1.1 Deductive logic

Given our classical logic system, we can also reason about propositions. To do so, we can make use of deductive logic. Deductive logic is a type of deduction where we take true statements (also called premises) and use these to form a conclusion. This can be illustrated in the following example:

$$\frac{\begin{array}{l}\text{We know that A is true}\\ \text{If A is true, B must also be true}\end{array}}{\text{Thus, B is true}}$$

Deductive logic systems contain a set of rules that indicate how premises can be transformed into conclusions. For instance, given a conjunction $A \land B$ as premise, it makes sense that we can conclude that A and B separately also must be true. In deductive logic, this intuition is defined by the following two rules.

$$\frac{\mathrm{A} \land \mathrm{B}}{\mathrm{A}} \qquad \frac{\mathrm{A} \land \mathrm{B}}{\mathrm{B}}$$

Likewise, the example introduced above can be defined using the following rule:

$$\frac{\mathrm{A} \quad \mathrm{A} \rightarrow \mathrm{B}}{\mathrm{B}}$$

2.2 Knights and knaves puzzles

Knights and knaves puzzles are puzzles in classical two-valued logic. There are persons, called knights, who can only tell the truth and others, called knaves, who can only lie. Nothing except their truth telling propensity tells them apart and the task is to figure out which person is a knight and which is a knave [12].

To make the idea of the puzzle a little more clear, we look at a classic example:

> We have two people: Person A and B. Person A says: "B and I are both knaves".

Now, the goal of this puzzle is to figure out whether person A and B are knights or knaves.

To solve this puzzle, we will start by assuming that A is telling the truth. If this was the case, it would mean that A is a knave. We assumed, however, that A was telling the truth, which cannot be the case for a knave. So, A has to be a knave and thus the statement they make cannot be correct. Following that logic, it cannot be the case that they are both knaves. Thus, we know that B has to be a knight and we have figured out this puzzle.

2.3 Three valued logic

In three valued logic there is a third truth value 'unknown' in addition to the classical truth values - true and false. This truth value indicates that there is vagueness in the statement that is being made, which means that it cannot either be verified or falsified.
We choose to preserve as much of the classical logic as possible. We let P_T indicate that P has the truth value true, P_F the value false, and P_U the value unknown. With some minor adjustments to fit the new third value, the Laws of Thought look as follows in three valued logic:

1. **The Law of Excluded Middle**
   For every proposition it holds that it is either true, false, or unknown
$$\forall P : P_T \vee P_F \vee P_U$$

2. **The Law of Identity**
   For every proposition it holds that it always retains to the same truth value
$$\forall P : P \rightarrow P$$

3. **The Law of Contradiction**
   For no proposition it holds that it is true, false, or unknown at the same time

$$\forall P : \neg(P_T \land P_F) \land \neg(P_T \land P_U) \land \neg(P_F \land P_U)$$

2.3.1 Logical operators

We can then create the truth tables for the basic logical operators - negation, conjunction, and disjunction - in three valued logic.

Table 2.2: Truth tables for negation, conjunction, and disjunction in three valued logic.

A	¬ A
T	F
F	T
U	U

A	B	A ∧ B
T	T	T
T	F	F
T	U	U
F	T	F
F	F	F
F	U	F
U	T	U
U	F	F
U	U	U

A	B	A ∨ B
T	T	T
T	F	T
T	U	T
F	T	T
F	F	F
F	U	U
U	T	T
U	F	U
U	U	U

We can compare Table 2.2 to Table 2.1 and see that for the classical truth values, the truth table of three valued logic is identical to the one in classical logic.

As mentioned before, we want to preserve as much of the classical way of reasoning as possible. Similar as in [4], we choose to interpret the value unknown as that it could possibly be true but also possibly be false to create these truth tables. This means that a conjunction returns true if both proposition are true and false if just one proposition is false. For disjunctions it holds that if just one proposition is true the whole statement returns true and if both propositions are false it returns false. Finally, it is intuitive that if the truth value of some statement is unknown, its negation will also be unknown.

In classical logic, the implication operator is defined as follows:

$$A \to B \equiv \neg A \lor B$$

Using this formula, we can create the truth table that is shown in Table 2.3. However, if we look closely at the entry for $U \to U$, we see that it is defined as T, even though $\neg U \vee U$ results in U. Our reason for doing this is so that we can preserve the Laws of Thought in our implementation of three valued logic. The Law of Identity namely states that it should hold that:

$$(A \to A) \equiv \texttt{True}$$

So, by changing $U \to U$ to evaluate to T, our system does not conflict with the Law of Identity anymore. This is an example of the many different choices that can be made among logic systems. Which choice is made should depend on what is useful for the eventual usage of this logic system. Later in Section 3.3.6 we will see that it is necessary to make another adjustment to the implication operator, in order for us to be able to correctly prove the puzzles.

For the bi-implication operator, we can make use of the following classical definition:

$$(A \leftrightarrow B) \equiv (A \to B) \wedge (B \to A)$$

Now that we have created a definition for both the implication and bi-implication operator, we can also create those truth tables.

Table 2.3: Truth tables for the original and adjusted implementation of the implication and bi-implication in three valued logic.

A	B	¬ A ∨ B
T	T	T
T	F	F
T	U	U
F	T	T
F	F	T
F	U	T
U	T	T
U	F	U
U	U	U

A	B	A → B
T	T	T
T	F	F
T	U	U
F	T	T
F	F	T
F	U	T
U	T	T
U	F	U
U	U	T

A	B	A ↔ B
T	T	T
T	F	F
T	U	U
F	T	F
F	F	T
F	U	U
U	T	U
U	F	U
U	U	T

2.4 Coq prover

Coq is a formal proof management system [6] that provides a formal language, which can be used to construct and check proofs. A Coq program consists of a series of Coq commands which are terminated by a period.

2.4.1 Types in Coq

Coq uses three different types internally.

- Prop is used for propositions.
- Set is used for computations.
- Type is the supertype of above mentioned types, meaning it allows programs to work with both Prop and Set.

2.4.2 Inductive types

The Inductive keyword allows us to create a new type from constants and functions that create terms of that type. For example, if we wanted to create the type bool as we know it, having the values true and false, we would write it down as follows:

```
Inductive bool : Type :=
  | true : bool
  | false : bool.
```

2.4.3 Definitions

To introduce functions in Coq we use the Definition keyword. We can supply the parameters of the function and specify what the return value should be. We can use the match function to pattern match on values.

```
Definition negation (b: bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

2.4.4 Notations

When implementing new functions, it might be convenient to provide infix notation symbols. This can simplify expressions that make use of many of these functions by quite a lot. If, for example, we have the function and that takes A and B as arguments, we can define it using the infix notation $\wedge$, to be able to use this symbol instead of the regular function application. Especially when we use multiple operators, the simplicity of infix operators becomes apparent. Instead of writing and A (and B (and C D)), we can simply write A /\ B /\ C /\ D.

Besides the symbol, we also need to specify the precedence level and the associativity of the operator. 100 is the lowest precedence level and 0 is the highest, meaning that operators with a precedence level closest to 0 are performed first.

In Coq, we can define new infix notation with the `Notation` keyword:

```
Notation "A /\ B" :=
    (and A B) (at level 80, right associativity).
```

2.4.5 Creating a Coq program

We always start by indicating the type of proof we are going to execute, which can be, for example, a `Theorem` or a `Lemma`. A `Theorem` is defined as a statement that has been proven to be true. A `Lemma` is similar to a `Theorem` in that it also is a true statement. The difference between the two is that in a `Theorem` we try to prove a certain end goal, while a `Lemma` is created specifically to help proving another `Theorem` or `Lemma`.

After that, we indicate the name of the program. Then, we indicate the variables that will be used in the environment and what type they are. The most important built-in type that we will work with is `Prop`, which stands for propositions. Finally, we enter the logical formula that we want to prove in this program.

```
Theorem example (p q : Prop): p -> q.
```

2.4.6 Executing the proof

When executing a proof we will make use of Coq commands that are called tactics. We can transform a certain state in the proof by transforming the goal or the hypotheses. This works in a similar way as we have seen in the deductive logic systems in Section 2.1.1: given a hypothesis or a goal of a certain form, we can apply a rule which transforms the formula accordingly. Through these transformations we should eventually get to a state where either our goal is in our hypotheses or there are two hypotheses that contradict each other. In both cases we can then prove the goal using the hypotheses.

We can look at the example of a tactic that is used to split a conjunction in the goal. If we use a deductive system, this tactic would look as follows:

$$\frac{A \qquad\qquad B}{A \wedge B}$$

There are a lot of different tactics. The most important ones, that we will also use the most, are the following:

apply	1. Applies an implication with a conclusion equal to the goal, and it transforms the goal into the premise of the implication 2. Applies a previously proved or assumed theorem, lemma or axiom
assumption	Solves the goal if it is equal to one of the hypotheses
contradiction	1. Solves the goal if there are two contradictory hypotheses 2. Solves the goal if one of the hypotheses is equal to `False`
destruct	1. Breaks a hypothesis consisting of a conjunction into two hypotheses 2. Breaks a hypothesis consisting of a disjunction into two subgoals with the same goal but different hypotheses 3. Breaks a hypothesis consisting of a bi-implication into two hypotheses
exact	Solves the goal if we know the exact hypothesis that solves it
intro	1. Transforms implications in the goal by moving the premise to the hypotheses and leaving only the conclusion as the goal 2. Introduces a new variable in place of a `forall` quantifier 3. When the goal is a negation, it moves the proposition within the negation to the hypotheses and transforms the goal to an inconsistency (`False`)
left	Transforms a goal consisting of a disjunction into a goal of the left side of the disjunction
reflexivity	Solves the goal if it is a true equality
right	Transforms a goal consisting of a disjunction into a goal of the right side of the disjunction
split	1. Breaks up a goal consisting of a conjunction into two separate subgoals 2. Breaks up a goal consisting of a bi-implication into two separate subgoals
tauto	Tries to execute a variety of tactics to automatically solve the proof

unfold	Unfolds the definition of the specified term
;	Applies the tactic on the right of the symbol to all the subgoals that are generated by the tactic on the left of the symbol

2.4.7 Ltac

Besides the standard tactics that are already implemented in Coq, we can also create our own tactics using the built-in system `Ltac`. The `match` tactic can be used to match hypotheses or goals with a specified pattern. We use question marks to indicate the metavariables, i.e., the variable parts in the hypothesis or goal that we want to give a name and might use in the tactic.

Before the `|-` symbol we can specify a hypothesis we want to match, and after the `|-` symbol we can specify a goal we want to match. Finally, we write the `=>` symbol, followed by the action that needs to be performed. This action could be a tactic we created ourselves, or one already known in Coq. The wildcard symbol `_` can be used to indicate that any value is allowed.

Two examples of the standard syntax of creating a tactic using match would look as follows.

```
Ltac example_tactic1 :=
  match goal with
    | [ H: ?a /\ ?b |- _ ] => destruct H
  end.

Ltac example_tactic2 :=
  match goal with
    | [ |- ?a /\ ?b ] => split
  end.
```

In the first example, we match our hypothesis with a formula of the form `?a /\ ?b`. This means that if the current hypothesis would be `P /\ Q`, then this would be of the correct form. We call the hypothesis that we match on `H` and then use the `destruct` tactic on this hypothesis `H`. The final result of `example_tactic1` will be the same as simply using the `destruct` tactic.

In the second example, we match our goal with a formula of the form `?a /\ ?b`. This means that a goal of the form `P /\ Q` is correct. We can then use the `split` tactic on this goal. The final result of `example_tactic2` will be the same as simply using the `split` tactic.

Chapter 3

Research

3.1 Knights and knave puzzles in a logical formula

3.1.1 Two valued logic

A knights and knave puzzle always consists of the same structure: there are people who are either knights or knaves, and these people might have one or more statements. When we rewrite a puzzle to a logical formula, we can represent all people in the puzzle as propositions. A person that is a knight would have truth value ‘true’, and a knave would have truth value ‘false‘. So if we have a puzzle with knight John in there, John could be presented as J. If John were a knave, however, then we would represent him as $\neg J$.

Moreover, the statements made by each person can also be represented using a propositional formula. And, since we know knights always speak the truth, we know that the propositional formula of a statement made by a knight must be true. Likewise, a statement made by a knave must always evaluate to false. As a result, we can state that the truth value of a person P must be equal to the truth value of the statement they made.

From these observations, a general structure can be found for rewriting knights and knave puzzles into a logical formula in two valued logic. We will explain this structure by means of an example of a puzzle.

> John says: "We are the same kind" and Bob says: "We are of different kinds".

We see that there are two people in the puzzle. We represent John as the proposition J and Bob as the proposition B.

Now we look at the first statement from John: "We are the same kind". If John is a knight this would mean that either John and Bob are both knights or they are both knaves. This can be written into a logical formula as $((J \land B) \lor (\neg J \land \neg B))$. This statement is, however, only true if (and only if) John is a knight. Hence, the fact that John makes this claim can be defined as the logical formula $J \leftrightarrow ((J \land B) \lor (\neg J \land \neg B)))$.

Here, we can see a general pattern for the conversion of a knight and knave puzzle to a logical formula. We take the claim of a person P_1 and convert it to a logical formula Q_1. Then, we assert that this formula is true if and only if that person is a knight, i.e., $P_1 \leftrightarrow Q_1$. We can combine all claims made by all people in a single logical formula:

$$\Phi = (P_1 \leftrightarrow Q_1) \land (P_2 \leftrightarrow Q_2) \land \cdots \land (P_n \leftrightarrow Q_n)$$

When reasoning in natural language, however, we generally approach these problems as a case distinction: what if John speaks the truth? And what if he lies? To be able to use this same methodology in our logical proofs, we can make use of the fact that a bi-implication can also be defined as a disjunction:

$$A \leftrightarrow B \equiv A \land B \lor \neg A \land \neg B$$

We look at the previous example again. There are two possibilities based on John's statement. Either John is a knight and it is true that he and Bob are of the same kind. The other possibility is that he is a knave, in which case his statement is not true. This can be formalized as the logical formula:

$$J \land (J \land B \lor \neg J \land \neg B) \lor \neg J \land \neg(J \land B \lor \neg J \land \neg B)$$

In a more general pattern, this becomes.

$$\Phi = (P_1 \land Q_1 \lor \neg P_1 \land \neg Q_1) \land \cdots \land (P_n \land Q_n \lor P_n \land Q_n)$$

Given a formalization of the statements, we want to try and obtain a conclusion to the problem. A solution is given as a combination of the roles of each person. For instance, a solution to the puzzle with John and Bob would be that John is a knave and Bob is a knight. In propositional logic, this can be defined as $\neg J \land B$. In general, we can formalize a proposed solution as the formula:

$$\Psi = \psi_1 \land \psi_2 \land \cdots \land \psi_n$$

Where $\psi_i = P_i$ if we think P_i is a knight, and $\psi_i = \neg P_i$ if we think P_i is a knave.

Of course, it could be the case that there is more than one solution to the puzzle. In that case, we encode the different solutions as a disjunction:

$$\Psi = (\psi_{i,1} \land \cdots \land \psi_{i,n}) \lor \cdots \lor (\psi_{j,1} \land \cdots \land \psi_{j,n})$$

Finally, we can define our entire puzzle as a logical problem where we take Φ to be our set of premises, and Ψ as our conclusions. Hence, we create a theorem:

$$\Phi \to \Psi$$

If our solution to the problem is correct, we can prove this logical problem using our set of deduction rules. Note that this system only works for puzzles for which we know the solution. This approach is meant to logically prove a solution to a puzzle, not to find all solutions to a given problem. For such a use case, it would be more useful to define the puzzle as a satisfiability problem, and use automated tools for solving such problems, as in [2].

3.1.2 Three valued logic

Since we only had to cover two types in two valued logic, we could make use of the fact that a variable was a knight and the negation of that variable meant it was a knave. In the variation with normal people, we can use the third value of our three valued logic system to represent a normal person. However, this also requires us to change the formalization of the statements. We can no longer use the bi-implication to indicate a statement is only true of the person saying it is a knight, as the person might also be a normal.

To formalize the variation correctly, we use the following definitions:

P_T: a person is a knight

P_F: a person is a knave

P_U: a person is unknown

If a person is unknown, this does not tell us anything about their statement. For the knight and knave we know that their claim will always be true for the former and false for the latter. Using all this information and the law of excluded middle, we can construct the following formula [11]:

$$(P_T \wedge Q) \vee (P_F \wedge \neg Q) \vee P_U$$

We can apply this formula to all people in the puzzle and their statements, and the conjunction of this is our set of premises Φ:

$$\Phi = (P_{1,T} \wedge Q_1 \vee P_{1,F} \wedge \neg Q_1 \vee P_{1,U}) \wedge \cdots \wedge (P_{n,T} \wedge Q_n \vee P_{n,F} \wedge \neg Q_n \vee P_{n,U})$$

Similar to our implementation in two valued logic, the conclusion Ψ of our puzzle is an assignment of roles to each person:

$$\Psi = \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n$$

Where ψ_i can be $P_{i,T}, P_{i,F}$ or $P_{i,U}$.

As in the two valued case, we can again formalize the entire puzzle as the formula:

$$\Phi \to \Psi$$

An example usage of this formalization can be found below.

Example 3.1.1. We look at one of the puzzle's out of the famous book of Smullyan *What is the name of this book?* [13]. The puzzle goes as follows:

> "A and B say the following:
>
> A: B is a knight.
> B: A is a knave.
>
> Prove that either one of them is telling the truth but is not a knight, or one of them is lying but is not a knave."

Using the previously defined formula and the statements as specified in the puzzle, we can construct the following formula as premise:

$$((A_T \wedge B_T) \vee (A_F \wedge \neg B_T) \vee A_U) \wedge$$

$$((B_T \wedge A_F) \vee (B_F \wedge \neg A_F) \vee B_U)$$

We have to prove that either A or B is of the type unknown, so our conclusion becomes:

$$A_U \vee B_U$$

This leaves us with the final formalization of the puzzle:

$$((A_T \wedge B_T) \vee (A_F \wedge \neg B_T) \vee A_U) \wedge$$

$$((B_T \wedge A_F) \vee (B_F \wedge \neg A_F) \vee B_U) \to$$

$$A_U \vee B_U$$

3.2 Knights and knave puzzles in two valued logic in Coq

3.2.1 Formalization of the puzzle

In Section 3.1.1 we have seen that we can define a proposition for every person which is true if the person is a knight and false if the person is a knave. The logical formula of a puzzle in natural language can then be constructed by following a set of patterns.

Example 3.2.1. We look at puzzle 33 out of the book of Smullyan [13]:

> Suppose A says, "I am a knave, but B isn't."
>
> What are A and B?

The solution to the puzzle is that A and B are both knaves.

The logical formula that we can create from this puzzle is as follows:

$$(A \wedge \neg A \wedge B) \vee (\neg A \wedge \neg(\neg A \wedge B)) \rightarrow$$

$$(\neg A \wedge \neg B)$$

We can turn this formula into the following Coq theorem:

```
Theorem smullyan_33 (A B: Prop):
  A /\ ~A /\ B \/ ~A /\ ~(~A /\ B) ->
    (~A /\ ~B).
```

When the puzzles get larger and more complex, and include more persons, the formulas using this structure also get really large. In order to improve readability, we create a function `formalize`, which looks as follows.

```
Definition formalize (person statement: Prop) :=
  person /\ statement \/ ~person /\ ~statement.
```

This function, given a person and a statement, creates a formula of the structure needed in our proof.

Example 3.2.2. Using this function, the theorem from the previous example would then become:

```
Theorem smullyan_33 (A B: Prop):
  formalize A (~A /\ B) ->
    (~A /\ ~B).
```

3.2.2 Executing the proof

We continue with the example from the previous section and will start the actual proof in Coq. In Section 2.4.6 we explain that `intro` can be used to move the premise of the implication to the hypothesis. We can choose the desired name for the hypothesis by putting the name behind it: `intro H`.

```
1 subgoal
A, B : Prop
H : A /\ ~ A /\ B \/ ~ A /\ ~ (~ A /\ B)
```

```
______________________________________(1/1)
~ A /\ ~ B
```

We now have a disjunction in the hypothesis that we can split using `destruct H as [H | H]`. This creates two cases with the same subgoal, but one has `A /\ ~ A /\ B` as hypothesis and the other `~ A /\ ~ (~ A /\ B)`. Since we see that both of these hypotheses are a conjunction we can use the operator `;` followed by the tactic `destruct H as [H1 H2]` to already split both of these cases.

```
2 subgoals
A, B : Prop
H1 : A
H2 : ~ A /\ B
______________________________________(1/2)
~ A /\ ~ B
______________________________________(2/2)
~ A /\ ~ B
```

We can already see there is a contradiction between `H1` and `H2`. We can use another `destruct` to separate the hypotheses in `H2` and solve the subgoal using `contradiction`.

```
1 subgoal
A, B : Prop
H1 : ~ A
H2 : ~ (~ A /\ B)
______________________________________(1/1)
~ A /\ ~ B
```

In `H2` we can see a common pattern in classical logic, namely De Morgan's Law:

$$\neg(P \land Q) \iff \neg P \lor \neg Q$$

Since the formula on the right side of the arrow is a lot easier to work with, we create and prove a lemma `morgan_law` that enables us to use De Morgan's Law in our proofs. By using the command `apply morgan_law in H2` we can transform the hypothesis to the alternate form.

```
1 subgoal
A, B : Prop
H1 : ~ A
H2 : ~ ~ A \/ ~ B
______________________________________(1/1)
~ A /\ ~ B
```

When splitting the disjunction using another `destruct`, we get two separate subgoals. One with `~ ~ A` and the other with `~ B` in our hypotheses. We can see immediately that we can solve the first case using `contradiction` and we continue with the second case.

```
1 subgoal
A, B : Prop
H1 : ~ A
H2 : ~ B
______________________________________(1/1)
~ A /\ ~ B
```

When using `split` we end up with two new subgoals, each containing one part of the conjunction. One of our subgoals thus contains `~ A` and the other `~ B`, while both of these values are also in our hypotheses. This means we can solve both these cases using `assumption` and thereby finish our proof. The full proof can be found in Appendix A.1.2.

We can see a pattern in that we evaluate all cases and either find a contradiction within the hypotheses or solve the goal using the hypotheses. We will show that this pattern is the same when executing a proof in three valued logic and explain it in more detail in Section 3.5.2.

3.3 Three valued logic in Coq

3.3.1 Three valued logic type

We introduce three valued logic into Coq by creating a new type that handles the three values it can take: true, false, and unknown. We call this new type `tvlBool`.

```
Inductive tvlBool : Type :=
  | true : tvlBool
  | false : tvlBool
  | unknown : tvlBool.
```

3.3.2 Logical operators

In Section 2, we introduced the truth tables for the logical operators negation, conjunction, and disjunction. We use the values in these truth tables and pattern matching on the inputs to implement these operators.

```
Definition tvlOr (x y: tvlBool) : tvlBool :=
  match x with
  | true => true
  | false => y
  | unknown => match y with
    | true => true
    | _ => unknown
    end
  end.
```

The implementation of the negation and conjunction operator are intuitively similar to the disjunction operator. The full implementation of all logical operators can be found in A.1.4.

For the bi-implication operator we don't make use of its truth table, but define it in terms of the implication operator. The implementation of the implication operator will be explained in a later subsection, but can be assumed to exist for now.

```
Definition tvlIff (x y: tvlBool): tvlBool :=
  tvlAnd (tvlImplies x y) (tvlImplies y x).
```

3.3.3 Validity of a formula

All operators we defined so far, evaluate to `tvlBool`. However, for a theorem to be valid in Coq it needs to evaluate to `Prop`. We introduce a new function `isValid` that evaluates to a `Prop` to solve this problem. `isValid` checks whether we are able to solve this logical formula, which is `True` if the formula returns `true` in three valued logic and `False` in the two other cases. Every formula in three valued logic that we try to prove will have to be evaluated by this function to give us a final result.

```
Definition isValid (x: tvlBool): Prop :=
  match x with
  | true => True
  | false => False
  | unknown => False
  end.
```

3.3.4 Notation

For convenience, instead of using the function names of each operator, we choose to use a custom symbolic notation to indicate the operators in three valued logic. We use the same symbols as in classical logic, but then prefixed with the # symbol to distinguish it from the classical symbols.

We use the same precedence and associativity as the corresponding classical operators that have been implemented in Coq already. This means that we start with the negation operator at a low level, and work our way up with the conjunction, disjunction, implication, and bi-implication operators.

All of the operators are defined to be right associative. Their full Coq implementation can be found in A.1.5.

Each function now corresponds to the following symbol:

Table 3.1: Coq function with their new notation.

Function	Symbol
`tvlNot x`	`#~x`
`tvlAnd x y`	`x #/\ y`
`tvlOr x y`	`x #\/ y`
`tvlImplies x y`	`x #-> y`
`tvlIff x y`	`x #<-> y`

3.3.5 Validity of three valued logic operators

Our next step would be to create tactics that work with three valued logic. In order to do so, however, we have to make use of the already existing tactics in Coq. The way we have defined our system now, we can create formulas like `isValid (a #/\ b)`, but the existing tactics in Coq are not able to handle three valued logic operators and the `isValid` wrappers out of the box.

To be able to modify our goals and hypotheses in order to advance our proof, we need to be able to rewrite them to a form that Coq's built-in tactics can handle. For instance, if we want to `split` a three valued logic conjunction, we first need to transform it to a regular conjunction. This can be done by showing the `isValid` operator is distributive over the operators. For instance, we can transform our conjunction `isValid (a #/\ b)` to a conjunction of Props `(isValid a /\ isValid b)`. This transformation, and the proof of this equivalence, is contained in the lemma `conjValid`.

```
Lemma conjValid:
  forall (a b: tvlBool),
    (isValid a /\ isValid b) <-> isValid (a #/\ b).
Proof.
induction a; induction b; unfold isValid; unfold tvlAnd;
tauto.
Qed.
```

We create similar theorems to handle the transformation for disjunction, implication, and bi-implication as well. The implementation of these three can be found in A.1.6.

3.3.6 Implication operator using the definiteness of a proposition

If we implement the implication according to the truth table from Section 2.3, we are not able to prove the lemma `impliesValid`. The lemma `impliesValid` is implemented as follows:

```
Lemma impliesValid:
  forall (a b: tvlBool),
    (isValid a -> isValid b) <-> isValid (a #-> b).
```

If we take the case where the truth value of `a` is `unknown` and the truth value of `b` is `false`, we can see why the lemma is incorrect. Below, we evaluate both sides of the equivalence to show why we are not able to successfully finish this proof.

```
isValid unknown -> isValid false <-> isValid (unknown #-> false)
                  False -> False <-> isValid (unknown)
                            True <-> False
```

It is obvious that this can never be the case in classical logic and we will, therefore, never be able to prove it in Coq. Since `impliesValid` is needed to be able to execute any tactic on an implication, we will have to change the truth table of the implies operator and define `tvlImplies` in a different way.

We make the choice to define the implication operator in terms of the definiteness of a proposition, as was also done in [3].

The definiteness of a proposition can be defined as follows:

$$\Delta(P) = \begin{cases} \texttt{false}, & \text{if } P = \texttt{unknown} \\ \texttt{true}, & \text{if } P = \texttt{true}, P = \texttt{false} \end{cases}$$

In Coq, we implement this as the function `isDefined`, by making use of pattern matching again.

```
Definition isDefined (x: tvlBool): Prop :=
  match x with
  | unknown => False
  | _ => True
end.
```

Now that we defined the definiteness of a proposition, we can define the implication operator in terms of it.

$$A \rightarrow B \;\equiv\; \neg\Delta(A) \vee \neg A \vee B$$

The truth table that follows from this is shown in Table 3.2. It is identical to Table 2.3, except for the case we were unable to prove. With this new definition, we re-define `tvlImplies` and are now successfully able to prove the lemma `impliesValid`. The new implementation of the implication can be found in Appendix A.1.4, and the proof for the lemma can be found in A.1.6.

Table 3.2: Adjusted truth table of the implication operator.

A	B	A $\rightarrow$ B
T	T	T
T	F	F
T	U	U
F	T	T
F	F	T
F	U	T
U	T	T
U	F	T
U	U	T

3.3.7 Tactics

Now that we have defined how to transform a formula using a three valued logic operator into a formula using a classical operator, we can start to define our own tactics. For every built-in tactic we have used in our proofs, we want to define a three valued logic variant.

We can implement these tactics in the way that has been explained in Section 2.4.7. To show this general pattern in practice, we look at the implementation of the three valued logic variant of the `intro` tactic.

Example 3.3.1. We start with the goal `isValid (a #-> b)`. The result we want after using `tvl_intro` is that `isValid a` is in the hypotheses and `isValid b` is in the goal.

We use `impliesValid`, as defined in Section 3.3.6, to transform `isValid (a #-> b)` into `(isValid a -> isValid b)`. The built-in tactic `intro` can be applied to a formula of this form and will result in `isValid a` in the hypothesis and `isValid b` as goal. This gives us the result we were looking for.

The implementation in Coq is as follows:

```
Ltac tvl_intro :=
  match goal with
  | |- (isValid (?a #-> ?b)) => apply impliesValid; intro
  end.
```

Most of the tactics can be defined in a similar fashion. The implementation of all our custom tactics can be found in A.1.8.

In the classical implementation of the `intro` tactic, we are able to choose whether we want to give a name to the hypothesis that is created or whether Coq should generate one. With the implementation of `tvl_intro` as seen in Example 3.3.1 we are currently not able to give our hypothesis a custom name. It is not possible to create two tactics with the same name, where one can handle an argument and the other cannot. Therefore, we create a new tactic `tvl_intro_base` in a way that it can take one argument: the name the introduced term should have. We can then create two different notations: `tvl_intro` without an argument and `tvl_intro` with an identifier as argument. If the first notation is used, we let Coq generate a fresh variable name and call `tvl_intro_base` with it as argument. If the second notation is used, we call `tvl_intro_base` with the name that was given as argument.

`tvl_intro` can now be used in the same way as in Example 3.3.1, but can now also handle it if the tactic is written like `tvl_intro P`. The implementation for the new `tvl_intro` tactic - limited to the introduction of implications - looks as follows:

```
Ltac tvl_intro_base name :=
  match goal with
  | |- (isValid (?a #-> ?b)) => apply impliesValid; intro name
  end.

Tactic Notation "tvl_intro" :=
    let H := fresh in tvl_intro_base H.
Tactic Notation "tvl_intro" ident(x) := tvl_intro_base x.
```

In the full implementation in A.1.8, we also support introductions of negations.

A similar problem occurs when we try to define `tvl_destruct`. In the classical implementation you can use `destruct H as [H1 | H1]` or `destruct H as [H1 H2]` to specify the names of the hypotheses generated from breaking up a conjunction or disjunction. The notation of `[H1 | H1]` and `[H1 H2]` are called ‘intro patterns’ in Coq. Intro patterns allow us to assign names to hypotheses that are introduced by using a tactic. We create a tactic `tvl_destruct_base` that lets us put the three valued formula in the correct form, i.e., which transforms conjunctions, disjunctions, and bi-implications into a form that is accepted by the built-in `destruct` tactic. Then, we create a variant of the `tvl_destruct` tactic that does not accept an intro pattern, which transforms the three valued formula and applies a `destruct` without pattern as well. The other variant, which does accept an intro pattern, also transforms the three valued formula, and then passes the intro pattern to the built-in `destruct` tactic.

Our final implementation of the `tvl_destruct` tactic looks as follows:

```
Ltac tvl_destruct_base H :=
  match type of H with
  | isValid (?a #/\ ?b) => apply conjValid in H
  | isValid (?a #\/ ?b) => apply disjValid in H
  | isValid (?a #<-> ?b) => apply iffValid in H
  end.

Tactic Notation "tvl_destruct" ident(x) :=
    tvl_destruct_base x; destruct x.
Tactic Notation "tvl_destruct" ident(x)
                 "as" simple_intropattern(pattern) :=
    tvl_destruct_base x; destruct x as pattern.
```

One limitation of this approach is that our `tvl_destruct` tactic does not support nested intro patterns, which the built-in `destruct` does support. Because it is not possible to inspect the intro pattern and apply the correct transformations to a formula, we accept that we can only destruct one operator at a time.

We should now be able to solve a logical formula with our three valued logic tactics to get to a result of the form `isValid a`, where `a` can be any value of `tvlBool` here, thus `true`, `false`, or `unknown`. To finish our proof, we need a tactic that applies the `isValid` function and gives us a `Prop` as result. The implementation of this tactic can be found below:

```
Ltac tvl_valid :=
  match goal with
  | |- (isValid true) => unfold isValid; trivial
  | _ => fail
  end.
```

3.4 Consequences of the adjusted implication operator

In Section 3.3.6 we made the choice to define the implication operator in Coq by making use of the definiteness of a proposition. The result of this was that `U #-> F` now evaluates to `true`, instead of `false` as it did with the original implementation. These types of choices usually have consequences for the properties that a certain operator has. We will evaluate the following properties related to the implication operator to see what effect this choice might have had:

Table 3.3: Properties of the implication operator.

Name	Formula
Contraposition	$P \to Q \equiv \neg Q \to \neg P$
Import-Export	$P \to (Q \to R) \equiv (P \wedge Q) \to R$
Negated conditionals	$\neg(P \to Q) \equiv P \wedge \neg Q$
Commutativity of antecedents	$(P \to (Q \to R)) \equiv (Q \to (P \to R))$
Distributivity	$(P \to (Q \to R)) \equiv ((P \to Q) \to (P \to R))$
Antecedent strengthening	$P \to Q \models (P \wedge R) \to Q$
Vacuous conditional	$\neg P \models P \to Q$
Transitivity	$(P \to Q) \wedge (Q \to R) \models P \to R$
Simplification of disjunctive antecedents	$(P \vee Q) \to R \models (P \to R) \wedge (Q \to R)$
Reflexivity	$\models P \to P$
Totality	$\models (P \to Q) \vee (Q \to P)$

All properties are evaluated by first creating a theorem in three valued logic in Coq that corresponds to the formula of the property. After this, we attempt to prove this formula. An overview of these proofs can be found in A.1.7.

All proofs can be executed without any problems, except the property of contradiction. We will look at the case that is not provable and thus makes the whole property incorrect:

```
isValid (true #-> unknown) = isValid (#~ unknown #-> #~ true)
          isValid (unknown) = isValid (unknown #-> false)
          isValid (unknown) = isValid (true)
                      False = True
```

It is obvious that this is indeed not provable and we can see it is caused by the definition choice we made for the implication operator.

Normally, this equivalence allows us to take a formula $P \to Q$, and prove it by 'proving the contrapositive' $\neg Q \to \neg P$. Since this property is not valid in our three valued logic system, a consequence of our choice for the implication operator is that we are not able to use this specific proof method in our proofs. However, in the remaining sections, we will see that the remaining equivalences are enough for us to be able to prove knights and knave puzzles in three valued logic.

3.5 Knights and knave puzzles in three valued logic in Coq

3.5.1 Formalization of the puzzle

In Section 3.1.2 we defined knights, knaves, and unknown people as P_T, P_F and P_U. When implementing these three values in Coq, we need to specify what these definitions mean exactly. We have created the function `isDefined` in Section 3.3.6 to indicate whether the variable has a known truth value in two valued logic, i.e., whether it is not unknown.

We can make use of the fact that the value `unknown` is not defined to create a definition for normal people. For knights and knaves, we know that `true` and `false` are both defined. We then combine that definiteness of the value with either the value itself or with the negation of the value, to check if it is `true` or `false` (i.e., a knight or a knave) respectively. This results in the following definitions in Coq:

```
Definition isKnight (x: tvlBool) := x #/\ isDefined x.
Definition isKnave (x: tvlBool) := #~ x #/\ isDefined x.
Definition isUnknown (x: tvlBool) := #~ isDefined x.
```

We can see that P_T would correspond to `isKnight P`, P_F to `isKnave P` and P_U to `isUnknown P`.

Example 3.5.1. In Section 3.1.2 we have seen the following logical formula for a knights and knaves puzzle:

$$((A_T \wedge B_T) \vee (A_F \wedge \neg B_T) \vee A_U) \wedge$$

$$((B_T \wedge A_F) \vee (B_F \wedge \neg A_F) \vee B_U) \to$$

$$A_U \vee B_U$$

We now have a definition for all three types and we already defined our logical operators in A.1.4. Therefore, we can easily translate the logical formula into a Coq program:

```
Theorem smullyan_41 (A B: tvlBool):
  isValid((isKnight A #/\ isKnight B #\/
            isKnave A #/\ #~isKnight B #\/
            isUnknown A)
      #/\ (isKnight B #/\ isKnave A #\/
            isKnave B #/\ #~isKnave A #\/
            isUnknown B)
   #-> (isUnknown A #\/ isUnknown B)).
```

Similarly as in classical logic, these formulas can get really large once the puzzles get more complex. We therefore also introduce a `formalize` function for three valued logic:

```
Definition formalize (person statement: tvlBool) :=
  isKnight person #/\ statement #\/
  isKnave person #/\ #~statement #\/
  isUnknown person.
```

Example 3.5.2. The formalization, as seen in the previous example, can then be transformed into the following theorem:

```
Theorem smullyan_41 (A B: tvlBool):
  isValid(
    formalize A (isKnight B) #/\
    formalize B (isKnave A) #->
    (isUnknown A #\/ isUnknown B)).
```

3.5.2 Executing the proof

Though we introduced the `formalize` function for readability of the logical formulas, we want to work with the underlying disjunctions whenever we execute the proof. Therefore, the first step in our proof will always be the tactic `unfold formalize`.

Example 3.5.3. We continue with the example from the previous section and first use `unfold formalize` to unfold the definition of `formalize` at all places it is used.

```
1 subgoal
A, B : tvlBool
______________________________________(1/1)
isValid
```

```
((isKnight A #/\ isKnight B
  #\/ isKnave A #/\ #~ isKnight B #\/ isUnknown A)
 #/\ (isKnight B #/\ isKnave A
     #\/ isKnave B #/\ #~ isKnave A #\/ isUnknown B)
 #-> isUnknown A #\/ isUnknown B)
```

In the rest of each proof, there is also a pattern that repeats itself. In Section 3.1.2 we have seen that the formalization of each puzzle follows a specific structure. When we start with our proof, the logical formula always consists of an implication from premise to conclusion:

$$\Phi \rightarrow \Psi$$

The proof starts with bringing the premise of this logical formula to the hypothesis of our Coq proof.

Example 3.5.4. To bring the left side of an implication to the hypothesis and leave the right side as goal, we make use of the `tvl_intro` tactic. We can specify what to call the new hypothesis, which we decide to call H. This leaves us with the final command `tvl_intro H`. This gives the following result in Coq:

```
1 subgoal
A, B : tvlBool
H : isValid((isKnight A #/\ isKnight B #\/ isKnave A
            #/\ #~ isKnight B #\/ isUnknown A)
          #/\ (isKnight B #/\ isKnave A #\/ isKnave B
              #/\ #~ isKnave A #\/ isUnknown B))
______________________________________(1/1)
isValid (isUnknown A #\/ isUnknown B)
```

Our premises are of the form:

$$\Phi = (P_{1,T} \wedge Q_1 \vee P_{1,L} \wedge \neg Q_1 \vee P_{1,U}) \wedge \cdots \wedge (P_{n,T} \wedge Q_n \vee P_{n,L} \wedge \neg Q_n \vee P_{n,U})$$

So we know the hypothesis consists of one or multiple statements connected with a conjunction. A conjunction in the hypothesis can be split up into smaller hypotheses for the same goal.

These smaller hypothesis are created as follows:

$$(P_T \wedge Q) \vee (P_F \wedge \neg Q) \vee P_U$$

Example 3.5.5. We continue with the previous example. A conjunction in the hypothesis can be split up into two hypotheses with the `tvl_destruct` tactic. We choose the names `H1` and `H2` for the new hypotheses. The full command we use is `tvl_destruct H as [H1 H2]` and gives us the following result in Coq:

```
1 subgoal
A, B : tvlBool
H1 : isValid (isKnight A #/\ isKnight B #\/ isKnave A
              #/\ #~ isKnight B #\/ isUnknown A)
H2 : isValid (isKnight B #/\ isKnave A #\/ isKnave B
              #/\ #~ isKnave A #\/ isUnknown B)
______________________________________(1/1)
isValid (isUnknown A #\/ isUnknown B)
```

Now that we have reduced our hypothesis into small workable hypotheses, we can start with the actual proof. To illustrate the further process of proving we will first look at the same example we have been using so far, but in natural language.

Example 3.5.6. A says: "B is a knight" and B says: "A is a knave". Prove that either one of them is telling the truth but is not a knight, or one of them is lying but is not a knave.

In natural language, we would start this proof by making an assumption about the type of A or B. For example, we could start with the assumption that A is a knight. From this assumption, it follows that what A says is true, meaning that it is correct that B is a knight. If B is a knight, it means that B is also telling the truth. From this, it follows that A should be a knave. However, we started with the assumption that A is a knight. So we can see that this is a contradiction and move on to the next case.

The hypotheses are currently a disjunction of different assumptions about the type of a person involved in the puzzle. We need to evaluate each of these cases and try to prove the goal with them. In order to evaluate each of these cases, we need to break up the disjunction which will create subgoals with different hypotheses for the same goal. If an assumption is not a correct one, according to the solution, we should be able to find a contradiction within the hypotheses. If an assumption is correct, we should be able to prove the goal with it.

Example 3.5.7. We start by breaking up the disjunction for the first hypothesis `H1`, which can be done using the `tvl_destruct` tactic. Since both hypotheses will be in different subgoals, we can let them keep the same name. This leaves us with the final command `tvl_destruct H1 as [H1 | H1]`, which gives the following result in Coq:

```
2 subgoals
A, B : tvlBool
H1 : isValid (isKnight A #/\ isKnight B)
H2 : isValid (isKnight B #/\ isKnave A #\/ isKnave B
              #/\ #~ isKnave A #\/ isUnknown B)
______________________________________(1/2)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(2/2)
isValid (isUnknown A #\/ isUnknown B)
```

The first case we will be evaluating is with the assumption that A is a knight, which is the same as in our example in natural language. We have already seen that this is not a correct solution and that we should be looking for a contradiction in the hypotheses. This foreknowledge is merely a convenience and is not necessary to be able to execute the proof.

If we assume that A is a knight, we thus assume its statement is true. So we can break this conjunction up into two hypotheses. To do this we use `tvl_destruct H1 as [H11 H12]`, which results in:

```
2 subgoals
A, B : tvlBool
H11 : isValid (isKnight A)
H12 : isValid (isKnight B)
H2 : isValid (isKnight B #/\ isKnave A #\/ isKnave B
              #/\ #~ isKnave A #\/ isUnknown B)
______________________________________(1/2)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(2/2)
isValid (isUnknown A #\/ isUnknown B)
```

Now, we need to look at where we can find a contradiction within these hypotheses. We split the disjunction in `H2` using the command `tvl_destruct H2 as [H2 | H2]`. Now we need to find a contradiction in this part of the hypothesis.

```
3 subgoals
A, B : tvlBool
```

```
H11 : isValid (isKnight A)
H12 : isValid (isKnight B)
H2 : isValid (isKnight B #/\ isKnave A)
______________________________________(1/3)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(2/3)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(3/3)
isValid (isUnknown A #\/ isUnknown B)
```

Then we use `tvl_destruct H2 as [H21 H22]` to split up the conjunction. This gives us one new assumption `isKnave A` and one we already assumed `isKnight B`.

```
3 subgoals
A, B : tvlBool
H11 : isValid (isKnight A)
H12, H21 : isValid (isKnight B)
H22 : isValid (isKnave A)
______________________________________(1/3)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(2/3)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(3/3)
isValid (isUnknown A #\/ isUnknown B)
```

By the Law of Contradiction we know that every person can only be of one type, so we see there's a contradiction between `isKnight A` and `isKnave A`. We unfold the definitions in these hypotheses using the commands: `unfold isKnight in H11` and `unfold isKnave in H22`. Our proof now looks like this:

```
3 subgoals
A, B : tvlBool
H11 : isValid (A #/\ isDefined A)
H12, H21 : isValid (isKnight B)
H22 : isValid (#~ A #/\ isDefined A)
______________________________________(1/3)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(2/3)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(3/3)
isValid (isUnknown A #\/ isUnknown B)
```

To be able to solve this case as a contradiction in Coq, we still first need to split up the hypotheses H11 and H22 into smaller cases. For this we use `tvl_destruct H11 as [H13 H14]` and `tvl_destruct H22 as [H23 H24]`. This gives us the following result:

```
3 subgoals
A, B : tvlBool
H13 : isValid A
H14 : isValid (isDefined A)
H12, H21 : isValid (isKnight B)
H23 : isValid (#~ A)
H24 : isValid (isDefined A)
______________________________________(1/3)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(2/3)
isValid (isUnknown A #\/ isUnknown B)
______________________________________(3/3)
isValid (isUnknown A #\/ isUnknown B)
```

Now, it is extremely clear there is a contradiction between H13 and H23, and we can use `tvl_contradiction` to solve this subgoal.

Now that we have proved the case in which we assume both A and B are knights, we can progress to the next case. Our hypothesis H2, which indicates our assumptions about B, now contains the following disjunction:

```
isKnave B #/\ #~ isKnave A #\/ isUnknown B
```

By applying `tvl_destruct H2 as [H2 | H2]` again, we can reduce this to two separate goals with different assumptions again. For these new subgoals, we will again try to prove our conclusion or find an inconsistency in our assumptions. By repeating this procedure over and over, we will be able to prove our goal for every combination of role for A and B.

The full proof of this puzzle can be found at A.1.10 in the Appendix.

In the previous examples, we have shown the general approach to solving a knights, knaves, and normals puzzle in three valued logic. The full proof for this puzzle - Puzzle 41 from Smullyan - can be found in Appendix A.1.10. We have also constructed a full proof for Smullyan 39, which can also be found in Appendix A.1.10. Its supporting functions can be found in A.1.9.

3.6 Solving puzzles with an incomplete solution

It can be the case that given a set of persons and statements, it is not possible to determine the type for each person.

An example out of the book of Smullyan is the following:

Example 3.6.1. Three of the inhabitants - A, B, and C - were standing together in a garden. A stranger passed by and asked A, "Are you a knight or a knave?" A answered, but rather indistinctly, so the stranger could not make out what he said. The stranger then asked B, "What did A say?" B replied, "A said that he is a knave." At this point the third man, C, said, "Don' t believe B; he is lying!"

The question is, what are A, B and C?

It is impossible for either a knight or a knave to say that they are a knave. Namely, if a knight would say he is a knave this is a false statement and similarly if a knave says he is a knave this is a true statement, both of which contradict the definitions of a knight and knave.

From this information, it follows that B must be lying and, therefore, is a knave. Then, from that we know that C told the truth and is a knight. In this case, there is too little information to determine what the type of A is.

Based on the information we do have, we can construct two different solutions: A is a knight, B is a knave, and C is a knight or A is a knave, B is a knave, and C is a knight. In a propositional formula, this becomes:

$$A \wedge \neg B \wedge C \vee A \wedge \neg B \wedge \neg C$$

Following the distributive laws, we can rewrite this as:

$$A \wedge \neg B \wedge (C \vee \neg C)$$

Since $C \vee \neg C$ is true by definition of the law of excluded middle, we can reduce the formula to:

$$A \wedge \neg B$$

So we can see that, if the type is not known for a person, we can leave this person out of our solution at all.

We can extend this into three valued logic in the same way. If we take as an example a puzzle that would have the same solution as the previous one (i.e., A is a knight and B is a knave), the following formula could be constructed:

$$A_T \wedge B_F \wedge C_T \vee A_T \wedge B_F \wedge C_F \vee A_T \wedge B_F \wedge C_U$$

We apply the same rules as in two valued logic to rewrite the formula to:

$$A_T \wedge B_F \wedge (C_T \vee C_F \vee C_U)$$

We see that also here, $C_T \vee C_F \vee C_U$ is true by definition of the law of excluded middle, and thus the final formula is:

$$A_T \wedge B_F$$

As a result, puzzles which have an incomplete solution (i.e., a solution in which the type of one or more people remains unknown) can simply be solved by leaving these people out of the proposed solution. The resulting solution is logically equivalent to the original disjunction of possible solutions, which means it can still be solved using the same deductive rules.

Chapter 4

Related Work

There exist different variations on how to approach the solving of a knights, knaves, and normals puzzle. For instance, in [9] the authors formalize knights and knaves puzzles using propositional logic, in a similar fashion to our formalization. Then, they use the consequence operator to generate extensions of this set of propositions, in order to verify whether a proposed solution does indeed follow from the formalization of the puzzle. Their approach differs from ours in the proof strategy, where we try to follow a specific pattern of deduction rules and they define their solution more theoretically in terms of the consequence operator.

In [2], a different approach using tableaux logic is introduced. In it, they formalize the puzzles using modal logic instead of the more classical propositional logic. They then add the negation of the conclusion of the puzzle to their assumptions, and try to disprove the resulting theorem by closing (i.e., finding inconsistencies in) each branch of the tableaux. Moreover, the author implemented their method in Prolog to support automated reasoning. As with the previous approach, this method uses a different method of proving the theorems than ours, i.e., one not grounded in deduction logic.

In [7] they introduce an implementation of three valued logic in Coq, which we have built upon. In addition to this implementation, we have described how to customize tactics to handle three valued logic and have shown that we are able to execute full proofs using these tactics. Other than this, we have found no other research on the implementation of a three valued logic system using a proof assistant. We have also not found any research implementing and proving knights and knaves puzzles in Coq.

Chapter 5

Conclusions

In this thesis, we have seen how we can formalize knights and knaves puzzles into a logical formula and how this formula can be used to prove the correctness of the solution of the puzzle. We have shown that there exists a general structure for both the formalization and the proof of these puzzles.

Moreover, we have constructed a three valued logic system by looking at the implementation of specific operators and the consequences of that implementation. To create an implementation in Coq, we also looked at the conversion between our three valued logic system and propositions as they are used in Coq.

Finally, we have shown that the general structure that is used to formalize knights and knaves puzzles can be extended to handle a variation of the puzzle, which includes normal people. We have seen that using the implementation of the three valued logic system in Coq, we are able to execute proofs with puzzles of this variation.

Chapter 6

Future work

6.1 Fuzzy logic

In natural language it is often not so clear whether a statement is fully true or false. Fuzzy logic, a concept introduced by Lofti Zadeh [15], handles this vagueness by giving every proposition a truth value ranging between 0 and 1. One indicates that the statement is completely true and zero indicates that it is completely false.

The following formulas for the logical operators can be constructed in the fuzzy logic system, where $v(P)$ indicates the value of a proposition P:

$$\begin{aligned} v(\neg P) &= 1 - v(P) \\ v(P \wedge Q) &= \min(v(P), v(Q)) \\ v(P \vee Q) &= \max(v(P), v(Q)) \end{aligned}$$

$$v(P \rightarrow Q) = \begin{cases} 1 & \text{if } v(P) \leq v(Q) \\ \text{1 - v(P) - v(Q)} & \text{if } v(P) > v(Q) \end{cases}$$

$$v(P \leftrightarrow Q) = v(P \rightarrow Q \wedge Q \leftrightarrow P)$$

A fuzzy variation of knights and knaves puzzles also exists, in which knights can transition to knaves and vice versa. However, this transition is not instant but rather a gradual process. As with fuzzy logic, we use values between 0 and 1 to represent where in the transition process a person is: someone with value 1 is a knight and someone with value 0 is a knave. If a person is transitioning but closer to being a knight (i.e., their value is larger than 0.5) we call them a *quasiknight*. Likewise, if a person is transitioning but closer to being a knave (i.e., their value is lower than 0.5) we call them a *quasiknave*. An extension of the research in this thesis could be to implement a fuzzy logic system in Coq and use this to represent and prove fuzzy knights and knaves puzzles.

It has been shown that the Laws of Thought, as introduced before, do not hold in fuzzy logic [10, 8]. Though it is possible to define the operators in such a way that it is possible to comply with these laws, this choice will result in other properties like distributivity and idempotence being violated [5]. When implementing fuzzy logic in future work, it is therefore also necessary to consider how to handle the Laws of Thought and what this means for the proofs.

6.2 Metapuzzles

Another variation of knights and knaves puzzles is introduced in Smullyan's book "The Lady or the Tiger" [14]. In these puzzles, which Smullyan refers to as 'metapuzzles', a puzzle is presented with insufficient information to solve it. It is given that someone else is able to solve it given certain additional pieces of information, though this extra information is withheld from the reader. The fact that someone is able to solve the puzzle given the withheld information provides a bit of partial information, which can be used to solve the puzzle. These puzzles can be constructed for regular knights and knaves puzzles, but also for the variations that include normal people.

An example puzzle out of "The Lady or the Tiger", for classical knights and knaves puzzles, is the following:

Example 6.2.1. A logician once visited this island and came across two inhabitants, A and B. He asked A, "Are both of you knights?" A answered either yes or no. The logician thought for a while, but did not yet have enough information to determine what they were. The logician then asked A, "Are you two of the same type?" A answered either yes or no, and the logician then knew what type each one was.

In this case, A will only answer 'no' to the first question if he is a knight and B is a knave. In all other cases the answer will be 'yes'. Since A's answer to the first question does not provide enough information to solve the puzzle, we know A must have answered 'yes'.

A's answer to the second question will be 'yes' if they are both knights, or if he is a knave and B is a knight. The answer will be 'no' if A is a knight and B is a knave, or if they are both knaves. The fact that A's answer leaves only one possible solution to the puzzle means that

A must have answered ‘no’. After all, the possibility that A is a knight and B is a knave was already ruled out by the previous question. Hence, the solution to this puzzle is that A and B are both knaves.

In this example, the piece of meta knowledge is whether or not the puzzle can be (uniquely) solved given a certain statement, without actually knowing this statement. However, it is difficult to define the ‘unique solvability’ of a puzzle in terms of the deductive operators we have introduced thus far.

So, as an addition to the formalization for a logical formula we currently have, we also want to be able to indicate that a unique solution can be found given the ‘meta knowledge’. Given all the statements as premise and the conclusion that there does indeed exist a unique solution, we should be able to prove the corresponding solution to the puzzle. Consequently, an extension to this thesis could be to look at the possibility of proving these metapuzzles in Coq, either using the current implementation of the three valued logic system, or by using a different implementation altogether.

Bibliography

[1] Richard B. Angell. *Reasoning and Logic.* New York: Appleton-Century-Crofts, 1964.

[2] L. Aszalós. A method to solve the puzzles of knights and knaves. *Publi. Math., Debrecen, kiozléesre beadva*, 2000.

[3] Arnon Avron. *Foundations and proof theory of 3-valued logics.* University of Edinburgh, Department of Computer Science, 1988.

[4] Jennifer Beineke and Jason Rosenhouse. *The Mathematics of Various Entertaining Subjects: Volume 3: The Magic of Mathematics*, volume 3. Princeton University Press, 2019.

[5] Radim Belohlavek, George J Klir, Harold W Lewis III, and Eileen C Way. Concepts and fuzzy sets: Misunderstandings, misconceptions, and oversights. *International journal of approximate reasoning*, 51(1):23–34, 2009.

[6] The Coq Team. The Coq proof assistant. `https://coq.inria.fr/`. Accessed at 19/12/2021.

[7] Emer Nı Dhomhnaill et al. Implementing a 3-valued logic in Coq.

[8] Gy. Fuhrmann. "Prototypes" and "fuzziness" in the logic of concepts. *Synthese*, 75(3):317–347, 1988.

[9] Adam Kolany. A general method of solving Smullyan's puzzles. *Logic and Logical Philosophy*, 4(4):97–103, 1996.

[10] George Lakoff. *Women, fire, and dangerous things: What categories reveal about the mind.* University of Chicago press, 2008.

[11] Jeff Pelletier. Using tableaux to solve knight-knave-normal problems, Oct 2011.

[12] Lance J. Rips. The psychology of knights and knaves. *Cognition*, 31(2):85–116, 1989.

[13] R.M. Smullyan. *What is the Name of this Book?: The Riddle of Dracula and Other Logical Puzzles.* Pelican books. Penguin Books, 1981.

[14] R.M. Smullyan. *The Lady or the Tiger? and Other Logic Puzzles.* Times Books, 1982.

[15] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

Appendix A

Appendix

A.1 Coq implementation

A.1.1 Auxiliary functions in two valued logic

```
Definition formalize (person statement: Prop) :=
  person /\ statement \/ ~person /\ ~statement.

Lemma morgan_law: forall P Q: Prop, ~(P /\ Q) <-> ~P \/ ~Q.
Proof.
intro P.
intro Q.
tauto.
Qed.

Lemma morgan_law2: forall P Q: Prop, ~(P \/ Q) <-> ~P /\ ~Q.
Proof.
intro P.
intro Q.
tauto.
Qed.
```

A.1.2 Proofs in two valued logic

```
Theorem smullyan_33 (P Q: Prop):
  formalize P (~P /\ Q) ->
  ~P /\ ~Q.
Proof.
unfold formalize.
intro H.
destruct H as [H | H]; destruct H as [H1 H2].
destruct H2 as [H2 H3].
contradiction.

apply morgan_law in H2.
destruct H2 as [H2 | H2].
contradiction.

split.
assumption.
assumption.
Qed.

Theorem smullyan_31 (P Q R: Prop):
  formalize P (~P /\ ~Q /\ ~R) /\
  formalize Q ((P /\ ~Q /\ ~R) \/ (~P /\ Q /\ ~R) \/
               (~P /\ ~Q /\ R)) ->
  ~P /\ Q /\ ~R.
Proof.
unfold formalize.
intro H.
destruct H as [H1 H2]; destruct H1 as [H1 | H1];
destruct H1 as [H1 H3]; destruct H2 as [H2 | H2];
destruct H2 as [H2 H4].
destruct H3 as [H31 H32].
contradiction.

destruct H3 as [H3 H5].
contradiction.

apply morgan_law in H3.
destruct H3 as [H3 | H3].
contradiction.

destruct H4 as [H4 | H4].
destruct H4 as [H4 H5].
contradiction.

destruct H4 as [H4 | H4].
destruct H4 as [H4 H5].
destruct H5 as [H5 H6].
split.
assumption.

split.
```

```
assumption.
assumption.

destruct H4 as [H4 H5].
destruct H5 as [H5 H6].
contradiction.

apply morgan_law in H3.
destruct H3 as [H3 | H3].
contradiction.

apply morgan_law in H3.
destruct H3 as [H3 | H3].
contradiction.

apply morgan_law2 in H4.
destruct H4 as [H4 H5].
apply morgan_law2 in H5.
destruct H5 as [H5 H6].
apply morgan_law in H6.
destruct H6 as [H6 | H6].
contradiction.

apply morgan_law in H6.
destruct H6 as [H6 | H6].
contradiction.
contradiction.
Qed.
```

A.1.3 Three valued logic type

```
Inductive tvlBool : Type :=
  | true : tvlBool
  | false : tvlBool
  | unknown : tvlBool.
```

A.1.4 Logical operators

```
Definition isValid (P: tvlBool): Prop :=
  match P with
  | true => True
  | false => False
  | unknown => False
  end.
```

```
Definition isDefined (P: tvlBool): tvlBool :=
  match P with
  | unknown => false
  | _ => true
end.

Definition tvlNot (P: tvlBool) : tvlBool :=
  match P with
  | true => false
  | false => true
  | unknown => unknown
  end.

Definition tvlAnd (P Q: tvlBool) : tvlBool :=
  match P with
  | true => Q
  | false => false
  | unknown => match Q with
    | false => false
    | _ => unknown
    end
  end.

Definition tvlOr (P Q: tvlBool) : tvlBool :=
  match P with
  | true => true
  | false => Q
  | unknown => match Q with
    | true => true
    | _ => unknown
    end
  end.

Definition tvlImplies (P Q: tvlBool) : tvlBool :=
  tvlOr (tvlNot (isDefined P)) (tvlOr (tvlNot P) Q).

Definition tvlIff (P Q: tvlBool): tvlBool :=
  tvlAnd (tvlImplies P Q) (tvlImplies Q P).
```

A.1.5 Notations

```
Notation "#~ P"
    := (tvlNot P) (at level 75, right associativity).
```

```
Notation "P #/\ Q"
    := (tvlAnd P Q) (at level 80, right associativity).

Notation "P #\/ Q"
    := (tvlOr P Q) (at level 85, right associativity).

Notation "P #-> Q"
    := (tvlImplies P Q) (at level 90, right associativity).

Notation "P #<-> Q"
    := (tvlIff P Q) (at level 95, right associativity).
```

A.1.6 Validity of three valued logic operators

```
Lemma negValid:
  forall (P: tvlBool),
    (isValid (#~ P)) <-> ~(isValid P) /\ isValid (isDefined P).
Proof.
intro a.
split.
intro H1.
split.
induction a; unfold tvlNot in H1; tauto.
induction a; unfold tvlNot in H1; unfold isValid in H1.
contradiction.
unfold isDefined.
tauto.
unfold isDefined.
tauto.
intro H2.
destruct H2.
induction a; unfold tvlNot; tauto.
Qed.

Lemma conjValid:
  forall (P Q: tvlBool),
    (isValid P /\ isValid Q) <-> isValid (P #/\ Q).
Proof.
induction P; induction Q; unfold isValid;
unfold tvlAnd; tauto.
Qed.

Lemma disjValid:
  forall (P Q: tvlBool),
    (isValid P \/ isValid Q) <-> isValid (P #\/ Q).
Proof.
induction P; induction Q; unfold isValid; unfold tvlOr; tauto.
Qed.

Lemma impliesValid:
  forall (P Q: tvlBool),
    (isValid P -> isValid Q) <-> isValid (P #-> Q).
Proof.
induction P; induction Q; unfold isValid; unfold tvlImplies;
unfold isDefined; unfold tvlOr; unfold tvlNot; tauto.
Qed.

Lemma iffValid:
  forall (P Q: tvlBool),
    (isValid P <-> isValid Q) <-> isValid (P #<-> Q).
Proof.
induction P; induction Q; unfold isValid; unfold tvlIff;
unfold tvlAnd; unfold tvlImplies; unfold isDefined;
unfold tvlOr; unfold tvlNot; tauto.
Qed.
```

A.1.7 Logical properties of the implication operator

```
Theorem importExport:
  forall (P Q R: tvlBool),
    isValid (P #-> (Q #-> R)) = isValid ((P #/\ Q) #-> R).
Proof.
induction P; induction Q; induction R; tauto.
Qed.

Theorem negConditionals:
  forall (P Q: tvlBool),
    isValid (#~ (P #-> Q)) = isValid (P #/\ #~ Q).
Proof.
induction P; induction Q; tauto.
Qed.

Theorem commutativity:
  forall (P Q R: tvlBool),
    isValid (P #-> (Q #-> R)) = isValid (Q #-> (P #-> R)).
Proof.
induction P; induction Q; induction R; tauto.
```

```
Qed.

Theorem distributivity:
  forall (P Q R: tvlBool),
    isValid (P #-> (Q #-> R)) =
    isValid ((P #-> Q) #-> (P #-> R)).
Proof.
induction P; induction Q; induction R; tauto.
Qed.

Theorem antecedent:
  forall (P Q R: tvlBool),
    isValid (P #-> Q) -> isValid ((P #/\ R) #-> Q).
Proof.
induction P; induction Q; induction R; unfold tvlAnd;
unfold tvlImplies; unfold tvlNot; unfold isDefined;
unfold tvlOr; unfold isValid; tauto.
Qed.

Theorem conditional:
  forall (P Q: tvlBool),
    isValid (#~ P) -> isValid (P #-> Q).
Proof.
induction P; induction Q; unfold tvlImplies; unfold tvlNot;
unfold tvlOr; unfold isDefined; unfold isValid; tauto.
Qed.

Theorem transitivity:
  forall (P Q: tvlBool),
    isValid ((P #-> Q) #/\ (Q #-> P)) -> isValid (P #-> Q).
Proof.
induction P; induction Q; unfold tvlAnd; unfold tvlImplies;
unfold tvlNot; unfold tvlOr; unfold isDefined;
unfold isValid; tauto.
Qed.

Theorem simplification:
  forall (P Q R: tvlBool),
    isValid ((P #\/ Q) #-> R) ->
    isValid ((P #-> R) #/\ (Q #-> R)).
Proof.
induction P; induction Q; induction R; unfold tvlAnd;
unfold tvlImplies; unfold tvlNot; unfold tvlOr;
unfold isDefined; unfold isValid; tauto.
```

```
Qed.

Theorem reflexivity:
  forall (P: tvlBool),
    isValid (P #-> P).
Proof.
induction P; unfold tvlImplies; unfold tvlNot; unfold tvlOr;
unfold isDefined; unfold isValid; tauto.
Qed.

Theorem totality:
  forall (P Q: tvlBool),
    isValid ((P #-> Q) #\/ (Q #-> P)).
Proof.
induction P; induction Q; unfold tvlImplies; unfold tvlNot;
unfold tvlOr; unfold isDefined; unfold isValid; tauto.
Qed.
```

A.1.8 Tactics

```
Ltac tvl_valid :=
  match goal with
  | |- (isValid ?P) =>
      match P with
      | true => unfold isValid; trivial
      | _ => fail
      end
  | _ => fail
  end.

Ltac tvl_left :=
  match goal with
  | |- (isValid (?P #\/ ?Q))
      => apply disjValid; left
  end.

Ltac tvl_right :=
  match goal with
  | |- (isValid (?P #\/ ?Q)) => apply disjValid; right
  end.

Ltac tvl_split :=
  match goal with
  | [ |- (isValid (?P #/\ ?Q))] => apply conjValid; split
  | |- (isValid (?P #<-> ?Q)) => apply iffValid; split
  end.

Ltac tvl_destruct_base H :=
  match type of H with
  | isValid (?P #/\ ?Q) => apply conjValid in H
  | isValid (?P #\/ ?Q) => apply disjValid in H
  | isValid (?P #<-> ?Q) => apply iffValid in H
  end.

Tactic Notation "tvl_destruct" ident(x) :=
  tvl_destruct_base x; destruct x.
Tactic Notation "tvl_destruct" ident(x)
               "as" simple_intropattern(pattern) :=
  tvl_destruct_base x; destruct x as pattern.

Ltac tvl_apply H :=
  match type of H with
  | isValid (?P #-> ?Q) => match goal with
    | |- (isValid Q) =>
        assert (forall (P Q: tvlBool),
          isValid (P #-> Q) -> (isValid P -> isValid Q)
        ) as Himpl;
        [>
          apply impliesValid |
          apply Himpl with P Q in H; [> assumption | clear Himpl]
        ]
    end
  end.

Ltac tvl_contradiction :=
  match goal with
  | [ H0 : (isValid (#~ ?P)), H1 : (isValid (?P)) |- _ ] =>
      apply negValid in H0; destruct H0; contradiction
  | [ H: (isValid false) |- _ ] =>
      unfold isValid in H; contradiction
  | [ H: (isValid unknown) |- _ ] =>
      unfold isValid in H; contradiction
  | [ H: (isValid (isDefined unknown)) |- _ ] =>
      unfold isDefined in H; unfold isValid in H; contradiction
  end.

Ltac tvl_intro_base name :=
  match goal with
  | |- (isValid (?P #-> ?Qb)) =>
       apply impliesValid; intro name
  | |- (isValid (#~ ?P)) =>
       apply negValid; split; [> intro name|]
  end.

Tactic Notation "tvl_intro" :=
    let H := fresh in tvl_intro_base H.
Tactic Notation "tvl_intro" ident(x) := tvl_intro_base x.
```

A.1.9 Auxiliary functions in three valued logic

```
Definition formalize (person statement: tvlBool) :=
  isKnight person #/\ statement #\/
  isKnave person #/\ #~statement #\/
  isUnknown person.

Lemma tvlDoubleNeg:
  forall (P: tvlBool),
    isValid (#~#~ P) <-> isValid (P).
Proof.
induction P; unfold tvlNot; tauto.
Qed.
```

A.1.10 Proofs in three valued logic

```
Theorem smullyan_41 (A B: tvlBool):
  isValid(formalize A (isKnight B) #/\
          formalize B (isKnave A) #->
          isUnknown A #\/ isUnknown B).
Proof.
unfold formalize.
tvl_intro H.
tvl_destruct H as [H1 H2].
tvl_destruct H1 as [H1 | H1].
tvl_destruct H1 as [H11 H12].
tvl_destruct H2 as [H2 | H2].
tvl_destruct H2 as [H21 H22].
unfold isKnight in H11.
unfold isKnave in H22.
tvl_destruct H11 as [H13 H14].
tvl_destruct H22 as [H23 H24].
tvl_contradiction.
```

```
tvl_destruct H2 as [H2 | H2].
tvl_destruct H2 as [H21 H22].
unfold isKnight in H12.
unfold isKnave in H21.
tvl_destruct H12 as [H13 H14].
tvl_destruct H21 as [H23 H24].
tvl_contradiction.

unfold isKnight in H12.
unfold isUnknown in H2.
tvl_destruct H12 as [H13 H14].
tvl_contradiction.

tvl_destruct H1 as [H1 | H1].
tvl_destruct H1 as [H11 H12].
tvl_destruct H2 as [H2 | H2].
tvl_destruct H2 as [H21 H22].
tvl_contradiction.

tvl_destruct H2 as [H2 | H2].
tvl_destruct H2 as [H21 H22].
tvl_contradiction.

tvl_right.
exact H2.
tvl_left.
exact H1.
Qed.

Theorem smullyan_39 (A B C: tvlBool):
  isValid (formalize A (isUnknown A) #/\
          formalize B (isUnknown A) #/\
          formalize C (#~ isUnknown C) #/\
          (isKnight A #\/ isKnight B #\/ isKnight C) #/\
          (isKnave A #\/ isKnave B #\/ isKnave C) #/\
          (isUnknown A #\/ isUnknown B #\/ isUnknown C) #->
          (isKnave A #/\ isUnknown B #/\ isKnight C)).
Proof.
unfold formalize.
tvl_intro H.
tvl_destruct H as [H1 H].
tvl_destruct H as [H2 H].
tvl_destruct H as [H3 H].
tvl_destruct H as [H4 H].
tvl_destruct H as [H5 H6].
tvl_destruct H1 as [H1 | H1].
tvl_destruct H1 as [H11 H12].
unfold isKnight in H11.
unfold isUnknown in H12.
tvl_destruct H11 as [H13 H14].
tvl_contradiction.

tvl_destruct H1 as [H1 | H1].
tvl_destruct H1 as [H11 H12].
tvl_destruct H2 as [H2 | H2].
tvl_destruct H2 as [H21 H22].
tvl_contradiction.

tvl_destruct H2 as [H2 | H2].
tvl_destruct H2 as [H21 H22].
tvl_destruct H4 as [H4 | H4].
unfold isKnave in H11.
unfold isKnight in H4.
tvl_destruct H11 as [H13 H14].
tvl_destruct H4 as [H41 H42].
tvl_contradiction.

tvl_destruct H4 as [H4 | H4].
unfold isKnave in H21.
unfold isKnight in H4.
tvl_destruct H21 as [H23 H24].
tvl_destruct H4 as [H41 H42].
tvl_contradiction.

tvl_destruct H6 as [H6 | H6].
tvl_contradiction.

tvl_destruct H6 as [H6 | H6].
unfold isKnave in H21.
unfold isUnknown in H6.
tvl_destruct H21 as [H23 H24].
tvl_contradiction.

unfold isKnight in H4.
unfold isUnknown in H6.
tvl_destruct H4 as [H41 H42].
tvl_contradiction.
```

```
tvl_destruct H3 as [H3 | H3].
tvl_destruct H3 as [H31 H32].
tvl_split.
exact H11.
tvl_split.
exact H2.
exact H31.

tvl_destruct H4 as [H4 | H4].
unfold isKnave in H11.
unfold isKnight in H4.
tvl_destruct H11 as [H13 H14].
tvl_destruct H4 as [H41 H42].
tvl_contradiction.

tvl_destruct H4 as [H4 | H4].
unfold isUnknown in H2.
unfold isKnight in H4.
tvl_destruct H4 as [H41 H42].
tvl_contradiction.

tvl_split.
exact H11.
tvl_split.
exact H2.
exact H4.

tvl_destruct H2 as [H2 | H2].
tvl_destruct H2 as [H21 H22].
tvl_destruct H5 as [H5 | H5].
unfold isUnknown in H22.
unfold isKnave in H5.
tvl_destruct H5 as [H51 H52].
tvl_contradiction.

tvl_destruct H5 as [H5 | H5].
unfold isKnight in H21.
unfold isKnave in H5.
tvl_destruct H21 as [H23 H24].
tvl_destruct H5 as [H51 H52].
tvl_contradiction.

tvl_destruct H3 as [H3 | H3].
tvl_destruct H3 as [H31 H32].
```

```
unfold isKnight in H31.
unfold isKnave in H5.
tvl_destruct H31 as [H33 H34].
tvl_destruct H5 as [H51 H52].
tvl_contradiction.

tvl_destruct H3 as [H3 | H3].
tvl_destruct H3 as [H31 H32].
apply tvlDoubleNeg with (isUnknown C) in H32.
unfold isKnave in H31.
unfold isUnknown in H32.
tvl_destruct H31 as [H33 H34].
tvl_contradiction.

unfold isUnknown in H3.
unfold isKnave in H5.
tvl_destruct H5 as [H51 H52].
tvl_contradiction.

tvl_destruct H2 as [H2 | H2].
tvl_destruct H2 as [H21 H22].
tvl_contradiction.

tvl_destruct H3 as [H3 | H3].
tvl_destruct H3 as [H31 H32].
tvl_destruct H5 as [H5 | H5].
unfold isUnknown in H1.
unfold isKnave in H5.
tvl_destruct H5 as [H51 H52].
tvl_contradiction.

tvl_destruct H5 as [H5 | H5].
unfold isUnknown in H2.
unfold isKnave in H5.
tvl_destruct H5 as [H51 H52].
tvl_contradiction.

unfold isKnight in H31.
unfold isKnave in H5.
tvl_destruct H31 as [H33 H34].
tvl_destruct H5 as [H51 H52].
tvl_contradiction.

tvl_destruct H3 as [H3 | H3].
tvl_destruct H3 as [H31 H32].
apply tvlDoubleNeg with (isUnknown C) in H32.
unfold isKnave in H31.
unfold isUnknown in H32.
tvl_destruct H31 as [H33 H34].
tvl_contradiction.

tvl_destruct H4 as [H4 | H4].
unfold isUnknown in H1.
unfold isKnight in H4.
tvl_destruct H4 as [H41 H42].
tvl_contradiction.

tvl_destruct H4 as [H4 | H4].
unfold isUnknown in H2.
unfold isKnight in H4.
tvl_destruct H4 as [H41 H42].
tvl_contradiction.

unfold isUnknown in H3.
unfold isKnight in H4.
tvl_destruct H4 as [H41 H42].
tvl_contradiction.
Qed.
```