Dissertation Plan

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

Draft: *January 20, 2025*

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *January 20, 2025*

Abstract

Abstract en

Keywords: One keyword, Another keyword, Yet another keyword, One keyword more, The last keyword

Resumo

Resumo pt

Palavras-chave: Primeira palavra-chave, Outra palavra-chave, Mais uma palavra-chave, A última palavra-chave

Contents

List of Figures

Acronyms

DAM Directed Acyclic Multigraph *v*, *14*, *15)*

FOL First-Order Logic *5–7*, *12*, *14*, *17*, *18)*

PL Propositional Logic *2–7*, *14)*

WFF Well Formed Formula *6)*

1 Introduction

Citing something online [**arxivIltisLearning**, **arxiv**].

This thesis will focus on the implementation and design of various types of exercises commonly found in introductory logic courses, both in PL and FOL. The system should be interactive, allowing students to automatically check their solutions, and, most importantly, it has to provide advanced assistance by delivering detailed feedback and hints without disclosing the correct answers. It also must provide an interface for teachers to add new exercises and automatically grade them.

1.1 Motivation

1.2 Problem formulation

1.3 Research

1.4 Document Structure

2

Background

2.1 Propositional Logic

Logic in general is defined as the study of the principles of reasoning. Propositional Logic (PL) is a branch of logic that focuses on the study of propositions and their relationships. The goal of logic in computer science is to create languages that help us represent situations we deal with as computer scientists. These languages allow us to think about and analyze these situations in a structured way. By using logic, we can build clear and valid arguments about these situations, ensuring they make sense and can be tested, defended, or even carried out by a machine [0].

Propositions are the basic building blocks of PL. A proposition is a declarative statement that has a truth value, which can be either true(denoted as T or 1) or false (denoted as F or 0), but not both.

Examples of Propositions:

- "Today is friday."
- "It is raining."
- "If it is cold, then it is raining."

2.1.1 Syntax

To define a formal language, one must choose the alphabet of the language and establish the set of words that make up the language. These words are usually called formulas when the formal language is associated with a logic, as is the case here. The alphabet of the language is a set of symbols, and each formula is a finite sequence of symbols from the alphabet.

Symbols are used to represent propositions and the relationships between them. By convention, propositions are represented by lowercase letters (p,q,r) or by Greek letters (ϕ,ψ,γ). The tables **??** and **??** list all the logical constants, propositional variables, and logical connectives in PL.

Symbol	Name	Example
$\top$	Top	$\top$: "True"
$\bot$	Bottom	$\bot$: "False"
p,q,r	Propositions	p: "It is raining."

Table 2.1: Logical Constants and Propositional Variables

Symbol	Name	Arity	Example
$\neg$	Not	1	$\neg p$: "It is not raining."
$\wedge$	And	2	$p \wedge q$: "It is raining and it is cold."
$\vee$	Or	2	$p \vee q$: "It is raining or it is cold."
$\rightarrow$	Implication	2	$p \rightarrow q$: "If it is raining, then it is cold."
$\leftrightarrow$	Equivalence	2	$p \leftrightarrow q$: "It is raining if and only if it is cold."

Table 2.2: Logical Connectives

A well-formed formula (WFF) in PL is defined recursively according to the following set of rules (**??**), which specify the conditions under which a formula is considered well-formed, and these rules build upon each other to allow for the construction of more complex logical expressions.

$$\begin{cases} \alpha \text{ is WFF} & \text{,if } \alpha \text{ is a proposition,} \\ \neg\alpha \text{ is a WFF} & \text{,if } \alpha \text{ is a WFF,} \\ (\alpha \wedge \beta) \text{ is a WFF} & \text{,if } \alpha \text{ and } \beta \text{ are WFFs,} \\ (\alpha \vee \beta) \text{ is a WFF} & \text{,if } \alpha \text{ and } \beta \text{ are WFFs,} \\ (\alpha \rightarrow \beta) \text{ is a WFF} & \text{,if } \alpha \text{ and } \beta \text{ are WFFs,} \\ (\alpha \leftrightarrow \beta) \text{ is a WFF} & \text{,if } \alpha \text{ and } \beta \text{ are WFFs} \end{cases}$$

Table 2.3: Rules for Well-Formed Formulas in Propositional Logic

Examples of WFF:

- $\top$: "True".
- $(p \wedge q) \rightarrow r$: "If it is raining and it is cold, then it is snowing."
- $(p \rightarrow q) \wedge (q \rightarrow r)$: "If it is raining, then it is cold, and if it is cold, then it is snowing."

2.1.2 Semantic

To define a language, we also need to define its semantics. In PL, this is not different, given a formula, we may want to determine its truth value. To do so, we must create an interpretation by assigning a truth value to each propositional symbol. An interpretation structure (V) over a set of propositional symbols P is a function $V : P \rightarrow \{0, 1\}$. We say that a formula (α) is satisfiable (SAT) over an interpretation ($V \models \alpha$) if the interpretation satisfies the formula (it evaluates true in that interpretation). Otherwise, the formula is not satisfiable ($V \not\models \alpha$). Table **??** presents the inductive definition of SAT in PL.

$$\begin{cases} \alpha \text{ is SAT} & \text{,if } \alpha \text{ is a proposition and } V(\alpha) = 1, \\ \neg\alpha \text{ is SAT} & \text{,if } \alpha \text{ is not SAT,} \\ (\alpha \vee \beta) \text{ is SAT} & \text{,if either } \alpha \text{ or } \beta \text{ is SAT,} \\ (\alpha \wedge \beta) \text{ is SAT} & \text{,if both } \alpha \text{ and } \beta \text{ are SAT,} \\ (\alpha \rightarrow \beta) \text{ is SAT} & \text{,if whenever } \alpha \text{ is SAT, then } \beta \text{ is SAT,} \\ (\alpha \leftrightarrow \beta) \text{ is SAT} & \text{,if both } \alpha \text{ and } \beta \text{ are either both SAT or both not SAT.} \end{cases}$$

Table 2.4: Inductive definition of SAT in PL

A formula is said to be possible if there is an interpretation that satisfies it. A more specific type of a possible formula is the one that is valid, or tautological, which is satisfied by any interpretation.

2.1.3 Equivalences

In PL, we may want to determine if two formulas are logically equivalent (≡), which occurs when, for all possible interpretations, both formulas always have the same truth values. There are several methods to check whether two logical formulas are equivalent:

- **Truth tables:** A truth table examines all possible interpretations for the propositions involved. To construct a truth table, we need to list all propositions, enumerate all possible truth value interpretations for those propositions, and evaluate the truth values of the formulas for each interpretations. If the results match in all rows, both formulas are equivalent. The number of interpretations can easily scale up depending on the number of propositions involved. For example, if we consider n propositions, the total number of rows will be 2^n. This is not a feasible method for evaluating large expressions due to the exponential growth in the number of rows. However, for smaller expressions, truth tables are effective. The table **??** demonstrates $p \rightarrow q \equiv \neg p \vee q$.

p	q	$p \rightarrow q$	$\neg p \vee q$
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

Table 2.5: Truth table showing the equivalence between $p \rightarrow q$ and $\neg p \vee q$.

- **Algebraic Manipulation:** This approach relies on a set of rules, known as logical equivalences, that can be used to manipulate expressions, similar to how it works in mathematics. By applying these equivalences, we can transform one logical expression into another that is logically equivalent, meaning both expressions have

the same truth value in every scenario. Here is the list of some well-known logical equivalences:

- **Identity Law:** $p \wedge \top \equiv p$ and $p \vee \bot \equiv p$
- **Domination Law:** $p \wedge \bot \equiv \bot$ and $p \vee \top \equiv \top$
- **Commutative Law:** $p \wedge q \equiv q \wedge p$
- **Associative Law:** $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
- **DeMorgan's Law:** $\neg(p \wedge q) \equiv (\neg p) \vee (\neg q)$ and $\neg(p \vee q) \equiv (\neg p) \wedge (\neg q)$

2.2 First-Order Logic

Another branch of logic is First-Order Logic (FOL), also known as predicate logic. Unlike PL, which focuses solely on simple declarative statements, first-order logic extends this by introducing quantifiers, predicates, variables, constants, and functions. These additional components allow us to express more complex declarative sentences, capturing relationships between objects and their properties in a specified context [0, 0].

Examples of First-Order Sentences:

- "There's a black cat that likes baths."
- "John is a friend of Mary."
- "If a variable is an integer and positive, then it is greater than zero."

2.2.1 Syntax

FOL uses the same syntax as PL but adds more features to make it more expressive. It introduces quantifiers that allow us to generalize or specify expressions, making it possible to express universal truths or existential statements. Quantifiers also enable the introduction of variables within a certain domain. Predicates that are used to express properties or relationships, allowing FOL to capture facts about objects and their interactions. They are always denoted with a capital letter, return a truth value, and can have different arities. Similar to predicates, FOL includes functions that represent mappings or computations. Functions are represented with lowercase letters, and their return value is a specific value in the domain.

Symbol	Name	Example
x, y, z	Variables	x: "An individual object"
$black$	Constants	$black$: "Value that is fixed"
$Cat(x)$	Predicates	$Cat(x)$: "True if x is a cat"
$color(x)$	Functions	$color(x)$ = black: "The color of x is black"

Table 2.6: Examples of variables, constants, predicates, and functions in FOL

Symbol	Name	Example
$\forall$	Universal	$\forall x\,(Cat(x) \rightarrow Mammal(x))$: "All cats are mammals"
$\exists$	Existential	$\exists x\,(Cat(x) \wedge color(x) = \text{black} \wedge LikesBaths(x))$: "There's a black cat that likes baths"

Table 2.7: Quantifiers in FOL

We can extend the definition of a Well Formed Formula (WFF) from PL to represent a WFF in FOL. To accomplish this, we must first introduce a new concept known as a term. A term is an expression that represents a specific value in the domain. The tables **??** and **??** show a simplified version of defining a WFF within FOL.

$$\begin{cases} c \text{ is a term} & \text{, if } c \text{ is a constant,} \\ x \text{ is a term} & \text{, if } x \text{ is a variable,} \\ f(t_1, t_2, \ldots, t_n) \text{ is a term} & \text{, if } f \text{ is a function with arity } n \text{ and } t_1, t_2, \ldots, t_n \text{ are terms.} \end{cases}$$

Table 2.8: Rules for Terms in FOL

$$\begin{cases} P(t_1, t_2, \ldots, t_n) \text{ is a WFF} & \text{, if } P \text{ is a predicate with arity } n \text{ and } t_1, t_2, \ldots, t_n \text{ are terms,} \\ \neg\alpha \text{ is a WFF} & \text{, if } \alpha \text{ is a WFF,} \\ \forall x\ \alpha \text{ is a WFF} & \text{, if } \alpha \text{ is a WFF and } x \text{ is a variable,} \\ \exists x\ \alpha \text{ is a WFF} & \text{, if } \alpha \text{ is a WFF and } x \text{ is a variable,} \\ (\alpha\ \{\wedge, \vee, \rightarrow, \leftrightarrow\}\ \beta) \text{ is a WFF} & \text{, if } \alpha \text{ and } \beta \text{ are WFFs.} \end{cases}$$

Table 2.9: Rules for Well-Formed Formulas in FOL

2.2.2 Semantic

As in PL, we must understand how semantics work in FOL. Unlike PL, in FOL, an interpretation structure is a pair ($M = (U, I)$). The first element is the domain (U), and the second is a function (I), that assigns an application to functions and predicates. Additionally, an assignment of variables X in M is a mapping $\rho : X \rightarrow U$, which associates each variable in X with an element of the universe U. Similarly to PL, we say that a formula (α) is SAT over an interpretation (M) and an assignment (ρ), denoted by $M, \rho \models \alpha$, if the interpretation satisfies the formula (it evaluates true in that interpretation). Otherwise, the formula is not satisfiable, denoted by $M, \rho \not\models \alpha$. Table **??** presents the inductive definition of SAT in FOL.

$$\begin{cases} \alpha \text{ is SAT} & \text{,if } \alpha \text{ is a predicate and } I(\alpha) = 1, \\ \neg\alpha \text{ is SAT} & \text{,if } \alpha \text{ is not SAT,} \\ (\alpha \vee \beta) \text{ is SAT} & \text{,if either } \alpha \text{ or } \beta \text{ is SAT,} \\ (\alpha \wedge \beta) \text{ is SAT} & \text{,if both } \alpha \text{ and } \beta \text{ are SAT,} \\ (\alpha \rightarrow \beta) \text{ is SAT} & \text{,if whenever } \alpha \text{ is SAT, then } \beta \text{ is SAT,} \\ (\alpha \leftrightarrow \beta) \text{ is SAT} & \text{,if both } \alpha \text{ and } \beta \text{ are either both SAT or both not SAT,} \\ \forall x\, \alpha \text{ is SAT} & \text{,if for every } u \in U, \alpha \text{ is SAT when assigning } u \text{ to } x, \\ \exists x\, \alpha \text{ is SAT} & \text{,if for some } u \in U, \alpha \text{ is SAT when assigning } u \text{ to } x. \end{cases}$$

Table 2.10: Inductive definition of SAT in FOL

2.2.3 Equivalences

Checking the equivalence of two expressions can be considerably more complicated in FOL compared to PL. FOL is way more expressive and is not easy to guaratee that for all interpretations in both expressions the truth value is the same. In fact, there is no algorithm that can prove the equivalence of all expressions. This problem is considered undecidable and was proven by Alan Turing [**turing1936**]. But there are some approaches that can be used to solve some kind of equivalences:

- **Contradiction:** To demonstrate by contradiction, we assume the negative of the expression and try to derive a contradiction from that assumption. If the assumption results in an absurdity, the original assertion must be valid.
- **Algebraic Manipulation:** This approach works the same way as the one described in PL. By applying these rules, we can transform one logical expression into another that is logically equivalent, meaning both expressions have the same truth value in every scenario. Here is the list of some well-known logical equivalences in FOL:
  - **Quantifier Negation:** $\neg\forall x P(x) \equiv \exists x \neg P(x)$ and $\neg\exists x P(x) \equiv \forall x \neg P(x)$
  - **Universal Distribution:** $\forall x(P(x) \wedge Q(x)) \equiv \forall x P(x) \wedge \forall x Q(x)$
  - **Existential Disjunction:** $\exists x(P(x) \vee Q(x)) \equiv \exists x P(x) \vee \exists x Q(x)$

2.3 Natural Deduction

A key task in Logic is to determine whether an expression is a semantic consequence of a set of formulas. Given a set of premises Φ and a conclusion ψ, it is said that ψ is a semantic consequence of Φ, denoted by $\Phi \models \psi$, if for every interpretation V over the set of propositions, if V satisfies all the formulas in Φ ($V \models \Phi$), then the interpretation V must satisfy ψ ($V \models \psi$). In other words, if the set of premises Φ is true, then ψ must also be true.

To determine if an expression is a semantic consequence of a set of formulas, one can use a deduction system. There are two ways to approach a deduction system: the semantic approach looks at what the formulas mean in different ways, and the syntactic

approach looks at how to use symbols in a deductive system [0]. There are numerous deductive systems in logic, some of the most well-known being resolution, tableau and natural deduction, but in this thesis we will concentrate on the natural deduction system.

Natural deduction is a type of syntactic deduction system that uses a pre-defined set of rules, called inference rules. By applying inference rules to the premises, we hope to get some more formulas, and by applying more inference rules to those, to eventually reach the conclusion [0].

There are many styles to represent these proofs. For example, the Fitch style uses a linear structure, with deeper indentation levels to represent assumptions or intermediate steps in the proof, while the tree style organizes the proof in a tree-shaped structure. In this thesis, we will focus on the tree-style representation.

These tree-shaped structures, also known as deduction trees, represent proofs and are built starting from individual trees and successively applying rules of inference. The rules are identified on the right-hand side of the fractions. There are 3 different types of rules: introduction (I), which creates more complex formulas from simpler ones; elimination (E), which does the opposite; and absurdity ($\bot$), which derives any conclusion from a contradiction. Table **??** presents some examples of inference rules.

$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \varphi & \psi\end{array}}{\varphi \wedge \psi}\ (\wedge I)$	$\dfrac{\begin{array}{c}\mathcal{D} \\ \varphi \wedge \psi\end{array}}{\varphi}\ (\wedge E_r)$	$\dfrac{\begin{array}{c}[\neg\varphi]^m \\ \mathcal{D} \\ \bot\end{array}}{\varphi}\ (\bot, m)$
Conjunction Introduction	**Conjunction Elimination, right**	**Absurdity**
$\dfrac{\begin{array}{c}[\varphi]^m \\ \mathcal{D} \\ \varphi\end{array}}{\varphi \to \psi}\ (\to I, m)$	$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \varphi & \neg\varphi\end{array}}{\bot}\ (\neg E)$	$\dfrac{\begin{array}{c}[\varphi]^m \\ \mathcal{D} \\ \bot\end{array}}{\neg\varphi}\ (\neg I, m)$
Implication Introduction	**Negation Elimination**	**Negation Introduction**

Table 2.11: Example of inference rules in Natural Deduction

Each rule has its own characteristic and can only be applied under certain circumstances. Some add new hypotheses that must be closed and others not. For instance, the Implication Introduction rule introduces a new hypothesis containing the antecedent of the implication, which can then be used to close the proof. Another example is the Conjunction Introduction rule that requires proving the left and right sides of the conjunction.

Individual trees are constructed from nodes, which are formulas. The formulas at the leaves are called hypotheses and are associated with marks (numbers). The formula at the root is the conclusion of the proof. Marks are used to identify the hypotheses that are given or derived from the rules. We call a closed hypothesis if its mark is used in a rule

of the tree otherwise, it is said to be open. The table below illustrates the procedure of a basic proof where we aim to prove: $\{\psi\} \models \varphi \rightarrow (\psi \wedge \varphi)$.

$$\frac{\psi \wedge \varphi}{\varphi \rightarrow (\psi \wedge \varphi)} \ (\rightarrow I, 2) \qquad \frac{\dfrac{\psi^1 \quad \varphi^2}{\psi \wedge \varphi} \ (\wedge I)}{\varphi \rightarrow (\psi \wedge \varphi)} \ (\rightarrow I, 2)$$

First step **Second step**

Table 2.12: Example of a deduction tree proving $\{\psi\} \models \varphi \rightarrow (\psi \wedge \varphi)$.

Building these proofs can be done in a variety of ways: bottom-up by starting from the conclusion and top-down by starting from the premises or open clauses. If we consider a bottom-up solution, the first step is to apply the Implication Introduction rule. From this rule, we derive a new hypothesis (φ), which we will mark with the number two, since we already have a premise marked with the number one. At this point in the proof, we have $\psi \wedge \varphi$ open. The second step is to apply the Conjunction Introduction rule, subsequently, both expressions in the leaves can now be closed with marks one and two, respectively. With no leaves left containing open hypotheses, the proof is declared complete when all necessary hypotheses have been closed and the desired conclusion is present at the root of the tree. So we proved $\{\psi\} \models \varphi \rightarrow (\psi \wedge \varphi)$.

In logic courses, students often struggle the most with these types of exercises. Some steps in the proofs are not immediately obvious, and trees can become quite large with many branches. Becoming proficient requires significant exposure and practice. The following schema illustrates a more complex example of a proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg(\varphi \vee \psi)^1 \quad \dfrac{\psi^2}{(\varphi \vee \psi)} \ (\vee I_l)}{\bot} \ (\neg E)}{\neg\psi} \ (\neg I, 2)}{\neg(\varphi \vee \psi) \rightarrow \neg\varphi} \ (\rightarrow I, 1)$$

Table 2.13: Example of a more complex deduction tree proving $\vdash \neg(\varphi \vee \psi) \rightarrow \neg\varphi$.

2.4 Proof Assistants

Proof assistants are software tools designed to help their users formalize programs or mathematical concepts and prove theorems about them [0]. Besides that, they can check step-by-step that the proof is correct according to a set of axioms and rules ensuring its correctness. Several proof assistants can also automate some steps, or even the full proof.

Libraries that provide reusable theorems, definitions, and strategies can extend them, enhancing efficiency and simplifying complex proofs.

Proof assistants can have a big impact in education, particularly for teaching mathematical reasoning and formal semantics. This type of tool can be used in Logic, for example, to help in constructing proofs in deduction systems. Some tools have a user-friendly interface, so the user can navigate through the steps of the proof to see the state on the step, can display information about the current goals, and provide little hints/suggestions about the steps to follow.

In the following sections, we present two examples of proof assistants that can be use in natural deduction.

2.4.1 Isabelle/HOL

Isabelle is a generic framework for interactive theorem proving. Isabelle/HOL is a large application within the generic framework that focuses on higher order logic (HOL). It includes a wide range of tools for logic-specific tasks and a large theory library [0, 0]. Isabelle/HOL is based on tactic functions that manipulate the proof state. These tactics can either solve a proof goal directly or break it into smaller subgoals. For instance, Blast is a first-order tableau prover, and Metis is a resolution prover.

Sledgehammer is an extremely powerful tool in Isabelle/HOL, which connects it with external provers by sending its problems to remote servers, increasing the efficiency of the prover. Additionally, it can automate proofs by utilizing various tactics that external provers have discovered. This automatization can be useful when combined with large proofs, as it can omit certain steps by using tactics. However, it may also hide some of the underlying reasoning behind the proof, making it harder for users to understand the intermediate steps. Since this tool cannot provide a full proof or a step-by-step resolution, it may not be suitable for developing our feedback system for natural deduction exercises.

However, Isabelle/HOL has tools for making counterexamples. For example, Nitpick uses a solver to systematically look for edge cases, and QuickCheck creates tests at random to test the properties of the expressions. These tools can be used in our feedback system to provide counterexamples to students, assisting them in identifying errors in their reasoning and improving their comprehension of the exercises. Figure **??** shows an example of how these tools are used and the corresponding counterexample found.

Figure 2.1: Example of code in Isabelle testing Nitpick and Quickcheck.

2.4.2 Lean 4

da

3

Related Work

In this section... TODO: Falta falar sobre MOOCs/MOOEPs. Explorar SAT solvers, o Coq e quickcheck quickchick (explorar formas de gerar provas de forma automatica)

3.1 Iltis Web-Based System for Teaching Logic

Iltis is an interactive online tool that assists students in learning logic from its foundation. [0, 0] The goal of this tool is to provide a system that supports a wide variety of content (propositional logic, modal logic, and first-order logic), along with a valuable feedback system that helps the learner better understand their mistakes. The developers of this web application divided it into multiple sections. Each section consists of a series of tasks, or exercises, that intensify in difficulty as the learner progresses through them. For each kind of task, this application provides a custom feedback generator. Feedback generators are pre-implemented pieces of code that dynamically provide various forms of feedback in tasks. This feedback can vary depending on the mistakes made by the learner. Some tasks have different levels of feedback that may differ based on the learner's proficiency. Low feedback levels provide a vaguer hint, and the high ones a more precise and explicit hint. Image **??** provides a list of the currently available types of exercises in Iltis.

Task	Description	Input	Output
Logical tasks			
PickVariable	Choose suitable propositional variables from a list.	–	variables $A_1, \ldots, A_m$
CreateFormula	Translate statements into formulas.	variables $A_1, \ldots, A_m$	formulas $\varphi_1, \ldots, \varphi_k$
InferenceFormula	Combine formulas $\varphi_1, \ldots, \varphi_k$ and a formula φ into a formula ψ that is unsatisfiable if and only if $\varphi_1, \ldots, \varphi_k$ imply φ.	formulas $\varphi_1, \ldots, \varphi_k$ and φ	a formula ψ
ManualTransformation	Textfield-based transformation of a formula φ into conjunctive, disjunctive or negation normal form, or into another formula.	a formula φ	the transformed formula ψ
GuiTransformation	Same as previous, but graphical user interface.	a formula φ	the transformed formula ψ
Resolution	Resolve the empty clause from the clauses of the CNF of φ.	a formula φ	–
Administrational tasks			
Questionaire task (ask a list of multiple choice questions), tasks to display messages, and a task to collect data and feedback from students.			

Figure 3.1: List of tasks available in Iltis.

From the teacher's perspective, this framework provides a way to create more tasks. Teachers can achieve this by creating an XML file where they specify a set of tasks and a list of feedback generators to be presented to the learner.

3.1.1 Feedback

Feedback generators comprise the Iltis feedback system. Teachers are allowed to associate more than one feedback generator with the task, creating different levels of assistance. Some exercises rely on feedback generators constructed using reversion rules, providing better and more accurate feedback. Reversion rules were built based on a previous study, where researchers collected some of the most frequent mistakes made by learners. A common example of a reversion rule in the "Propositional Formulas" exercises is to switch the order of the antecedent and consequent in implications. Whenever a learner switches two parameters, the feedback generator tries to apply reversion rules to find the correct solution. If successful, this indicates that the solution is close to the correct one, making it possible to provide more precise feedback based on the applied rule(s). Otherwise, it suggests that the solution is far from the correct one.

3.1.2 Conclusion

There are some positive aspects to consider from this system when developing our own tool, such as the intuitive way (it presents a low learning curve, and it is fundamental for these kinds of tools) that the exercises are presented to the learner, the advanced feedback system, and the simple access to the tool. It also provides a vast set of exercise types and a modular way to create them. On the other hand, teachers need to specify tasks in XML, and this requires some extra knowledge. Some types of exercises are still missing in this tool, like the deduction tree proof. Since it was developed by a German university and is not open-source, it cannot be expanded.

3.2 Logic4Fun

Logic4Fun is an online tool with a wide range of logical problems and puzzles focused on logical modeling and formalization ([0]).

This tool has been under development since 2001 by an Australian university. It was projected to help students practice and develop skills in formalizing logical problems, as this is a challenging topic to teach, and learners often struggle with it.

Logic4Fun uses many-sorted first-order logic (MSFO) [1] language to express the problems. It has a solver that takes as input a set of formulas and searches for finite models of this set. This tool presents a web page with different levels of problems: Beginner, Intermediate, Advanced, Expert, and Logician, with increasing complexity. It starts with trivial exercises to help students better understand how to use the site (declare vocabulary, set constraints, and read the solver output) and progresses to more complex and challenging exercises that require a strong background in logic. One of the key advantages of

[1] Many-sorted first-order logic is an extension of FOL. In FOL, all variables come from the same domain, limiting flexibility when modeling exercises with multiple distinct domains. MSFO extends this by allowing the assignment of types (or sorts) to variables and predicates, making the language more expressive.

using this tool is the ability to receive immediate and accurate feedback, in contrast with traditional teaching methods. This helps keep learners motivated and encourages them to invest more time and effort into solving problems. This site also allows users to enroll in a course by using the credentials provided by the teacher.

3.2.1 Feedback

Logic4Fun tracks two kinds of errors: syntactic and semantic. Syntactic errors are mistakes in the structure or arrangement of words that violate the grammar rules of the language. These errors can be captured by the parser or type checker. When a user attempts to submit an exercise with syntactic errors, a message is presented with some suggestions. When the type checker detects an error, it provides more information, especially about the expected and given types. Semantic errors are mistakes in logic or meaning in a programming language that occur in program execution. Since there are no predefined solutions, these errors are harder to classify and to deal with. Given these difficulties, Logic4fun created a diagnostic tool to provide more informative responses to the users when a solution cannot be found. The diagnostic tool uses two approaches to provide information: approximate models and unsatisfiable cores.

- In the approximate models approach, the solver starts by marking some unsatisfied constraints as "soft" and then attempts to satisfy as many as possible. Then a user can adjust constraints and rerun the solver, iterating to find the optimal approximations.

- In the unsatisfiable cores approach, the solver can try to identify groups of unsatisfied constraints that are causing the problem. Each group must contain at least one contradiction, giving useful clues for troubleshooting the problem.

3.2.2 Conclusion

Logic4Fun has several positive aspects, such as allowing exercises to be saved, enabling learners to pause their work and resume it later, progressively increasing the difficulty of exercises, helping learners integrate with the tool, and incorporating a diagnostic tool to address the lack of feedback. It has a class system where professors can invite their students to enroll. However, it only provides a restricted range of exercises based on first-order logic. It has some limitations with the solver's performance (the number of models presented to the user is restricted), and it is still facing issues with feedback. Sometimes, the reported errors are overly detailed or unclear, which can become frustrating for the learners.

3.3 LOGAX

LOGAX is a tool designed to help students in constructing Hilbert-style axiomatic proofs[2]. in PL ([0]). This tool is capable of providing feedback and hints at different levels of the proofs. It can give the solutions for the steps to follow, as well as the complete solution to the problem.

The team behind LOGAX focused on various methods to provide better assistance to users during the exercises. One of the methods that they found entails the teacher providing a hidden solution or deriving it from a set of student solutions. However, this method has some drawbacks: it can only recognize solutions that are nearly identical to the stored proofs, it is limited to a fixed set of exercises, and every time a teacher wants to add a new exercise, they must provide a hidden solution. Another method that they found relies on Bolotov's algorithm, which addresses all the issues of the previous method.

Bolotov's algorithm is a proof searching algorithm for the natural deduction in FOL. For any given problem, if solvable, the algorithm terminates, either finding a corresponding natural deduction proof or giving a set of constraints, from which a counter-example can be extracted ([0]).

In natural deduction, students are not strictly required to solve the exercises in one way. Proofs to the same problem can have different shapes, use different rules/axioms, and the order of the steps that the users follow can be different (**??**). LOGAX tool has a powerful adaptation of this algorithm that covers all the drawbacks previously mentioned in the hidden solution approach, as well as the flexible characteristics of solving Natural Deduction proofs.

LOGAX's algorithm can adapt its solution space to better assist learners based on the user's steps. If the user takes a step in the current solution space, the algorithm can give feedback and hints directly. However, if the step diverges from the solution space, it is necessary to recalculate it, and then the system uses the new solution space to compute feedback and hints. This dynamic behavior enables the system to consistently provide feedback and hints to the user, preventing them from becoming lost in the exercise. To make this happen, the algorithm creates a directed acyclic multigraph (Directed Acyclic Multigraph (DAM)) that can hold more than one possible answer to a given proof. A better description of the algorithm and its adaptations can be seen in [0, 0]. In this DAM, vertices represent statements, and edges connect dependent statements (rule application).

Image **??** shows an example of generating a DAM, where three assumptions are given (p, $p \to q$ and $q \to r$) and the goal is to prove $q \to r \vdash (p \to q) \to (p \to r)$. There are solid red edges that show how to use Modus Ponens (which is the same thing as the Implication Elimination rule) and dashed blue edges that show how to use the Deduction Theorem (which is the same thing as the Implication Introduction rule). The statements $(q \to r) \to (p \to (q \to r))$ and $p \to (q \to r) \to ((p \to q) \to (p \to r))$ are axioms. In this

[2]Axiomatic proofs are a kind of proof in formal deduction where each step of the proof is supported by axioms or inference rules previously established.

example the DAM captures three different solutions for the same proof: one that uses axioms a and b, one that uses the Deduction Theorem and axiom a and one that uses no axioms and applies the Deduction Theorem twice.

Storing this information in a DAM makes it easier to provide feedback about the steps required to complete the proof at any given level. The user can apply the rules in any order they choose. The system can adapt to the user's solution if it diverges, simply by following the sequence of rules chosen by the user. This allows the system to easily provide information about next steps (top-down proof) or previous steps (bottom-up proof) using the edges. To provide a complete solution to the user, it is necessary to extract and trim the solutions from the DAM.

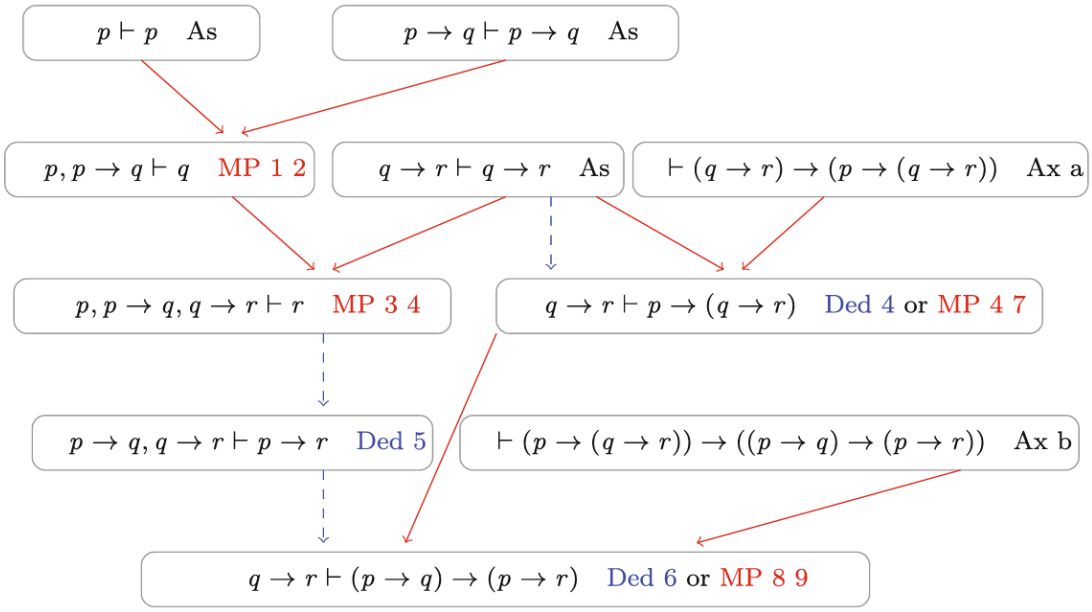

Figure 3.2: Example of a DAM generated by LOGAX.

LOGAX's team not only focused on the feedback but also the way they presented the exercises to the users. The design of this tool focuses on interfaces that allow students to concentrate on the goal of solving proofs rather than waste time figuring out syntactic errors. Shortening the number of steps and distractions required to reach the goal leads to better learning outcomes.

3.3.1 Hints & Feedback

The LOGAX algorithm primarily covers the feedback and hints system. It can provide the directions for the next steps, indicate the next rule to be applied, and offer an explicit step-by-step procedure for performing the next step. To make the assistance system even more powerful, the team extended it to provide not only informative support but also information about subgoals. For example, rather than trying to directly prove the conclusion, the helping system offers hints in the form of sub-proofs that lead step-by-step

to the final proof. By providing feedback that includes information about subgoals, the system can help students understand why a certain step is useful, and students are more likely to succeed in correcting mistakes.

The team also did some studies on students' common mistakes during proofs, and they came up with the following types of mistakes:

- Oversights: Correspond to syntactic errors, for example, when a user forgets to close a parenthesis in a sentence or when logical symbols are not placed in the correct position.

- Conceptual errors: These occur when a student fundamentally misunderstands a concept or its application. For example, this error can occur when choosing the statements to apply a certain rule.

- Creative rule adaptations: These occur when students try to invent their own rules. For example, from this $\neg p \rightarrow (q \rightarrow \neg r)$ and q, we can conclude $\neg p \rightarrow \neg r$ using Modus Ponens (Implication Elimination rule). This usually happens when the student does not know how to proceed.

The helping system underwent a final adaptation to track these mistakes. This way, the system can point out a mistake and, if possible, mention exactly which formula, subformula, or set of formulas does not match the chosen rule.

3.3.2 Conclusion

For the deduction tree exercises (REF deduc tree exercises), LOGAX appears to be a perfect fit. It covers a wide range of aspects to consider when developing a system like this. Starting with the algorithm that finds multiple possible solutions for a given proof, the fact that it ignores the order of the steps, and the ability to adjust the guidance based on the user's solution. It also includes different approaches for giving feedback and hints, as well as the idea of giving subgoals as hints to help the student understand the proof. Additionally, there are some important aspects regarding the design of the interfaces.

Unfortunately, this is an old project, and the tool is no longer available. This project has some minor drawbacks that we can consider when developing our solution. One of them is that the system doesn't provide fading strategies to reduce the amount of feedback. We might want to control the amount of feedback sent to the student based on their level of expertise. Another problem this tool faces is that the user can't erase lines of the proof. As a consequence, the final proof can be more extensive than expected. Overall, it seems to meet all the requirements for a good tutoring system, and for sure, this tool will be used as a reference for the one that we are going to develop.

3.4 MineFOL

MineFOL is a game for learning First Order Logic ([0]). This game is very similar to a well-known game called Minesweeper. Minesweeper is a game that features a grid containing empty cells and cells with hidden mines. The object is to locate all the hidden mines using hints provided as the player explores the grid. These hints indicate the number of mines adjacent to a given cell, helping the player deduce their locations. MineFOL is an adaptation of that game where, instead of giving hints with the number of the nearest mines, it gives messages with FOL expressions to help locate them, for example: $\neg\exists x\,\text{mine}(1, x)$. These expressions are hidden in safe squares, and the goal is to find the maximum number of expressions to have enough information to locate all mines in as few steps as possible. The player can only travel through safe squares. A square is considered safe if it is possible to prove it using the collected messages. If the player steps out of the proven zone, the game ends. The game always starts at the top left corner of the grid($cell(1, 1)$) with an initial expression. Players always have information about how many mines there are and the maximum number of moves to solve the problem. We can combine this information with the collected expressions to extract more valuable insights. Image **??** illustrates a game where the objective is to locate one mine within 35 movements.

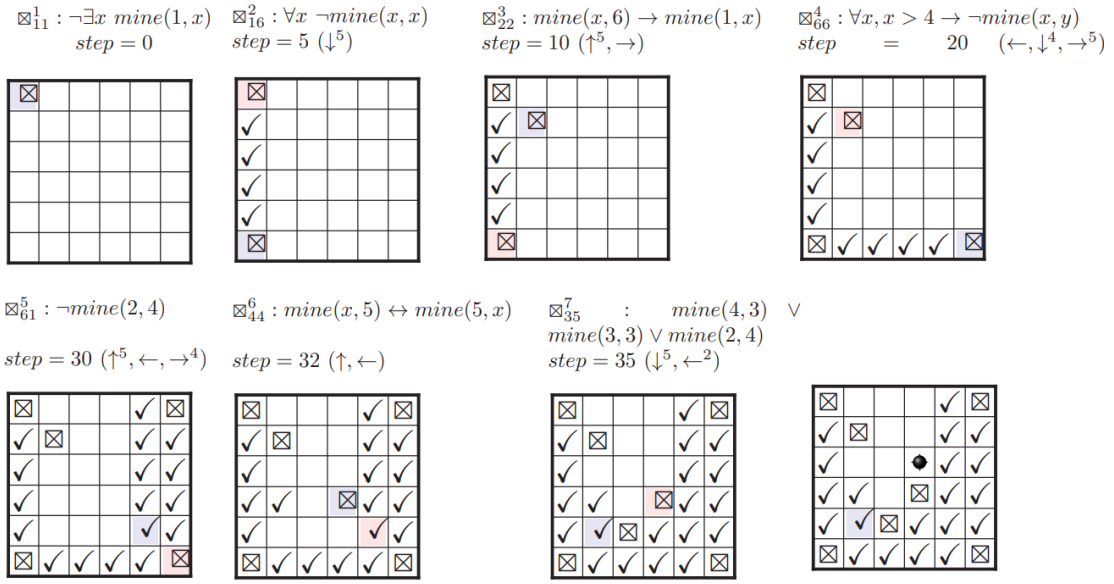

Figure 3.3: MineFOL example with one mine and 35 movements. Blue squares represent the player's current position, and the red ones represent the previous position. Cells with: ⊠ contain a message and ✓ represent safe squares.

MineFOL has three different modes:

- "Play it yourself!": In this mode the user tries to locate all mines based on a set of discovered FOL expressions. This can help the student practice reasoning in FOL,

since the student has to translate the expressions to natural language to understand the message in order to solve the game.

- "Challenge a software agent!": In this mode, the computer is responsible for solving the game. The student doesn't need to worry about anything. If the game is valid, the agent will find a solution for it, and they will present it to the student.
- "Create your own game!": In this mode, the user is responsible for setting up the game. Students can define their own FOL expressions, as well as the size of the grid and the number of mines. The complexity of the game can vary based on the grid size, the number of mines, the size and complexity of the FOL expressions, and the number of steps required to complete it. This mode is a good way for students to practice formalization in FOL. To check if the game is valid (can be solved), the student can challenge the software agent.

3.4.1 Feedback

This game includes a feedback system that provides assistance every time the player loses a game. When a player leaves the safe zone, the system provides an explanation of why the cell is unsafe by displaying a resolution-based proof.

3.4.2 Conclusion

MineFOL introduces a new concept that none of the previously described tools has. The concept of gamification consists of applying game mechanics and concepts in non-gaming environments. Studies show that if the learning platform is gamified, it does not only drastically increase the user enrollment but also increase user engagement throughout the course ([0]). Using MineFOL, students develop skills in reasoning and formalization while playing the game. It also allows competitiveness and creativity between students by creating and challenging other students to complete their own maps. The game also has different levels of complexity, allowing students with less experience to have a chance to learn, while more advanced students can challenge themselves with harder levels to further improve their skills.

4

Proposed Work

The aim of this thesis is to design and implement a variety of types of exercises [REF] commonly found in introductory logic courses, both in PL and FOL. To enhance the students' engagement, the exercises should be presented in an interactive way. This should be supported by a feedback system that will guide students through the resolution of the exercises and help prevent them from getting lost. Developing a successful feedback system is not an easy task. We need to find a balanced way to provide the right amount of feedback without leaving the student even more lost in the exercise. This can depend on many factors, for example, the level of expertise of the student, the type of exercise, the resolution path that the student is considering, etc. We described some excellent examples of feedback systems in [CHAPTER 3]. This tool must also provide teachers with an intuitive way to add new exercises as well as a way to grade them. The purpose of this is to enable automatic evaluation through integration with existing online e-learning platforms like Moodle.

4.1 Exercises

There is a vast variety of exercises in logic. Below is a list of some exercises, each one followed with a brief description of how they work.

- **Transforming a sentence from natural language to PL/FOL:** This type of exercise essentially involves translating a declarative sentence in natural language into PL/FOL. For example, consider the following propositions:

  $$p : \text{It is raining}, \quad q : \text{It is cold}.$$

  Now, imagine the exercise asks you to write "If it is cold then it is raining" in PL, using the propositions defined above. A correct answer would be: $q \rightarrow p$ or $\neg q \vee p$. While developing this exercise, it is essential to consider that a question may have multiple answers, so an equivalence checking system is needed. Implementing an equivalence checking system seems pretty straightforward in PL, but when dealing

with FOL expressions, the problem turns out to be extremely hard, since there's no algorithm that can always prove the equivalence of two different expressions[REF].

- **Build Truth Tables:** This exercise involves filling in the gaps of a truth table. The user is presented with an expression in PL, and the goal is to determine the final truth values by considering all possible combinations of truth values for the propositions. To simplify the exercise, the user can break the initial formula into smaller subformulas and, at the end, combine their values to compute the final truth value.

- **Conversion to Conjunctive, Disjunctive, and Negation Normal Forms:** In this exercise, students receive an expression and must transform it into various forms. For example, consider this expression:

  $$(\varphi \wedge \psi) \vee \neg\theta$$

  The exercise asks you to convert this expression into CNF by distributing disjunctions over conjunctions, resulting in: $(\varphi \vee \neg\theta) \wedge (\psi \vee \neg\theta)$. Alternatively, converting to DNF involves distributing conjunctions over disjunctions, resulting in: $(\varphi \wedge \psi) \vee \neg\theta$.

- **Natural deduction in tree shape:** In this exercise, students are required to prove the validity of an expression both in PL and FOL via natural deduction. As described in [REF], these proofs are based on established rules that can be used to infer new results that will help reach a conclusion. A detailed step-by-step resolution of these kinds of exercises is provided in [REF]. This is the type of exercise where students struggle the most, so it would be important to consider it when designing our system.

4.1.1 Ambition Levels

4.1.2 Evaluation Plan

4.2 Work Plan

Bibliography

[0] J. Blanchette, L. Bulwahn, and T. Nipkow. *Automatic Proof and Disproof in Isabelle/HOL*. (Visited on 2025-01-20).

[0] A. Bolotov et al. *Automated First Order Natural Deduction. Automated First Order Natural Deduction*. 2005. (Visited on 2024-12-12).

[0] G. Geck et al. *Iltis: Learning Logic in the Web*. (Visited on 2024-11-15).

[0] G. Geck et al. "Introduction to Iltis: an interactive, web-based system for teaching logic". In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (2018-07), pp. 141–146. DOI: 10.1145/3197091.3197095. (Visited on 2024-11-15).

[0] P. Gouveia and F. Dionísio. *Lógica Computacional Capítulo 1 -Lógica Proposicional*. (Visited on 2025-01-13).

[0] A. Groza, M. Baltatescu, and M. Pomarlan. *MineFOL: a Game for Learning First Order Logic*. (Visited on 2025-01-13).

[0] M. Huth and M. Ryan. *Logic in Computer Science*. 2004. (Visited on 2024-10-25).

[0] J. Lodder et al. "Generation and Use of Hints and Feedback in a Hilbert-Style Axiomatic Proof Tutor". In: *International Journal of Artificial Intelligence in Education* 31 (2020-11), pp. 99–133. DOI: 10.1007/s40593-020-00222-2. (Visited on 2024-12-01).

[0] A. Schlichtkrull. "Formalization of Algorithms and Logical Inference Systems in Proof Assistants". In: *Scandinavian Conference on AI* (2015-01), pp. 188–190. DOI: 10.3233/978-1-61499-589-0-188. URL: https://www.semanticscholar.org/paper/Formalization-of-Algorithms-and-Logical-Inference-Schlichtkrull/86a3bb90de4bfb3eff46ef3dd86967f869d53d9d (visited on 2025-01-19).

[0] J. Slaney. *Logic for Fun: an online tool for logical modelling*. (Visited on 2024-11-20).

[0] A. Vaibhav and P. Gupta. *Gamification of MOOCs for Increasing User Engagement*. (Visited on 2025-01-13).

[0] M. Wenzel, L. Paulson, and T. Nipkow. *The Isabelle Framework*. (Visited on 2025-01-20).