Media Engineering and Technology Faculty
German University in Cairo

A Comprehensive Study on Anti-unification and Implementation of a Visual Tool

Bachelor Thesis

Author:	Shahd Hesham Shehata
Supervisors:	Dr. Ahmed M. H. ABDELFATTAH
Submission Date:	29 May 2025

Media Engineering and Technology Faculty
German University in Cairo

A Comprehensive Study on Anti-unification and Implementation of a Visual Tool

Bachelor Thesis

Author:	Shahd Hesham Shehata
Supervisors:	Dr. Ahmed M. H. ABDELFATTAH
Submission Date:	29 May 2025

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor's Degree

(ii) due acknowledgment has been made in the text to all other material used

Shahd Hesham
29 May 2025

Acknowledgments

I thank...

Abstract

Anti-unification is a process that seeks to identify the most general common generalization between two symbolic expressions, playing a vital role in various AI fields such as machine learning, case-based reasoning, cognitive modeling, and knowledge discovery. This thesis investigates the theoretical principles of anti-unification and proposes the development of an interactive tool aimed at helping users visualize and explore the process of anti-unification. The tool will be built using existing open-source libraries that implement algorithms for anti-unification across multiple symbolic expression theories, including first- and second-order terms, higher-order patterns, and nominal terms. The thesis also explores ways to enhance these algorithms to improve the tool's performance and adaptability. In addition, it examines how the tool could be integrated with other AI applications, such as machine learning systems and cognitive reasoning platforms, to support generalization tasks. The developed tool will feature a user-friendly interface that visualizes the anti-unification process, compares various algorithms, and provides detailed, interactive explanations of the generalization steps. This project aims to contribute a valuable resource to the study and application of anti-unification in AI by combining theoretical analysis with practical software development.

Contents

List of Figures

Comments, Feedback, and Hints:

1. Dear Shahd, I remember we've agreed to write something that is your own work and reflects your understanding. Moreover, a lot of what we agreed on adding is still missing. This will affect your progress! Please revise your writings and produce a better thesis that reflects your progress AND your understanding.

2. Moreover, where is anything related to the practical part?

3. In addition, where is ANY kind of scientific citation. This is bad; take care!!!

4. This very recent paper is a very good candidate to be summarized and linked to the theoretical, scientific research part of your thesis. It should be your homework to summarize the paper and cite the related parts. Some parts of the paper can even be utilized in the practical part, if you can see the appropriate linking between your work and the results presented and discussed in the paper.

   You should add the Bib entry in the bibliography file (cite as: [1]), and I have uploaded the .PDF to my Google shared folder to everyone at `https://drive.google.com/drive/folders/15S7NZXr9A6ysxj7Pu94nzmgoyjVreXTT?usp=drive_link`.

```
@misc{WhyMASfail2025,
  title={Why Do Multi-Agent LLM Systems Fail?},
  author={Mert Cemri and Melissa Z. Pan and Shuyi Yang and Lakshya
    A. Agrawal and Bhavya Chopra and Rishabh Tiwari and Kurt Keutzer
     and Aditya Parameswaran and Dan Klein and Kannan Ramchandran
    and Matei Zaharia and Joseph E. Gonzalez and Ion Stoica},
  year={2025},
  eprint={2503.13657},
  archivePrefix={arXiv},
  primaryClass={cs.AI},
  url={https://arxiv.org/abs/2503.13657},
}
```

Reply:

1. Dear Doctor, I already added my understanding in the topic like for example i wrote what i read about plotkin and gave an example the example part the tree is just an example is it wrong to put examples?like im just visualizing what plotkin meant. where exactly is the part missing it?

2. The practical part im still working on it so there is no final words to put in the thesis yet like the code for example im working on building it to include all the things i mentioned that it will include

3. The citation i included it in the Biblo.bib file should i also added it in the paragraph including the research

4. Also doctor you will find CODEFORDOCTOR.tex at the end of the files here is my code so far the one im working on but cant include in the thesis because its not final yet

Chapter 1

Introduction

1.1 Insights (unsorted)

Figure 1.1 shows the responsible part of the code that does bla bla bla. Visit this page

Bringing clarity to complexity, this thesis presents an interactive tool that visualizes anti-unification, making it easier to explore how AI systems find common patterns and drive smarter generalizations

1.2 Motivation

Anti-unification plays a crucial role in AI, enabling generalization in tasks like machine learning, reasoning, and knowledge discovery. However, existing methods, such as Plotkin's anti-unification, primarily focus on syntactic generalization and often overlook deeper semantic similarities. Many traditional approaches are computationally expensive, limiting their practicality in large-scale applications. Furthermore, first-order logic (FOL) and propositional logic (PL) present unique challenges, such as handling function symbols, predicates, and formula simplifications. Addressing these limitations can improve automated reasoning and AI-driven knowledge extraction, making anti-unification more efficient and applicable to real-world problems.

1.3 Problem Statement

Despite its importance, current anti-unification techniques struggle with handling theoretical variations, particularly in first-order and propositional logic. In first-order logic, existing approaches fail to generalize predicates and functions effectively, often producing overly broad or overly specific results. The lack of a widely accepted strategy for generalizing function symbols and handling quantified expressions further complicates

```python
def draw_tree(expression, title="Expression Tree"):
    """Draws a tree representation of an expression using NetworkX."""
    if expression is None:
        return  # Skip if input is invalid

    G = nx.DiGraph()  # Create a directed graph

    def add_nodes_edges(expr, parent=None):
        """Recursively add nodes and edges for visualization."""
        if isinstance(expr, tuple):  # If it's a function
            node_label = expr[0]  # Function name (e.g., 'f')
            node_id = f"{node_label}_{id(expr)}"  # Unique ID
            G.add_node(node_id, label=node_label)

            if parent:
                G.add_edge(parent, node_id)  # Connect to parent

            for sub_expr in expr[1:]:  # Process arguments
                add_nodes_edges(sub_expr, node_id)
        else:
            G.add_node(expr, label=expr)  # Base case: variable or constant
            if parent:
                G.add_edge(parent, expr)  # Connect to parent

    add_nodes_edges(expression)

    # Draw the graph
    pos = nx.spring_layout(G)  # Positions for nodes
    labels = {node: G.nodes[node]['label'] for node in G.nodes}  # Node labels

    plt.figure(figsize=(6, 4))  # Set figure size
    nx.draw(G, pos, with_labels=True, labels=labels, node_size=2000, node_color="lightblue", font_size=10, edge_color="gray")
    plt.title(title)
    plt.show()
```

Figure 1.1: Part of the code given at sdfkjsdkfjksdf, showing sdkfhksdfjkfdjkfj

the process. In propositional logic, current methods do not efficiently incorporate formula simplification techniques, leading to redundant generalizations that fail to recognize structurally different but logically equivalent expressions. This research aims to develop a novel anti-unification framework that overcomes these challenges by refining generalization rules, integrating logic-aware techniques, and optimizing structural abstraction while maintaining logical correctness.

1.4 Objectives

The objectives of this work are:

- First: To enhance Plotkin's anti-unification algorithm by addressing its limitations in handling first-order logic (FOL) and propositional logic (PL), focusing on improving the treatment of function symbols, predicates, and formula simplification.
- Second: To develop an interactive visual tool that demonstrates the improved anti-unification process, allowing users to visualize and explore generalizations across various symbolic expression types.

1.5 Thesis Outline

In this thesis, we will explore anti-unification, focusing on improving existing methods and developing a visual tool to aid in understanding the process. In 1 we introduce the fundamental concepts of anti-unification, its significance, and its applications in artificial intelligence. The second chapter 2.1 presents the theoretical foundations necessary for understanding anti-unification. It covers first- and second-order logic, existing generalization techniques, and relevant mathematical concepts. In Chapter 3 we will be introducing additional content or refining discussions as the research progresses. Chapter 4 is how the project was developed,

- **Concept Overview:** Briefly explains anti-unification, first- and second-order logic, and the technologies used in the project.
- **Literature Review:** Summarizes previous work on anti-unification, existing algorithms, and related research. Identifies limitations and gaps.
- **Methodology:** Outlines the workflow, improvements to Plotkin's anti-unification, and the steps to develop the visual tool.
- **Results:**
  - **Dataset Comparison:** Evaluates different datasets used for testing.

  - **Project Results:** Analyzes the tool's performance and improvements over existing methods.

- **Conclusion:** Summarizes key findings, contributions, and proposes future enhancements to the tool and algorithm.

Chapter 2

Background

2.1 Concepts Overview

This chapter provides an overview of key concepts related to anti-unification, its applications, and its challenges.

2.1.1 Anti-Unification

Anti-unification is the process of finding the most general common abstraction between two symbolic expressions. This process plays a crucial role in various artificial intelligence fields, including machine learning, cognitive modeling, automated theorem proving, and knowledge representation.

In essence, anti-unification seeks to identify the shared structure of two expressions while replacing their differing parts with variables. The resulting generalized expression captures a pattern that can apply to multiple instances.

Example

Consider the two expressions:

$$f(a, b) \tag{2.1}$$

$$f(a, c) \tag{2.2}$$

Here, the structure of both expressions is the same, except for the second argument (b in the first expression and c in the second). Anti-unification replaces the differing parts with a variable to obtain the most general form:

$$f(a, X) \tag{2.3}$$

where X represents the unknown or generalized part.

2.1.2 First- and Second-Order Logic

First-Order Logic (FOL)

First-order logic (FOL) is a formal system used to represent relationships between objects. It consists of:

- **Variables:** (e.g., x, y)
- **Constants:** (e.g., a, b, c)
- **Functions:** (e.g., $f(x), g(y)$)
- **Predicates:** (e.g., $P(x, y), Q(x)$)
- **Logical Operators:** $(\wedge, \vee, \neg, \rightarrow)$
- **Quantifiers:** ($\forall$ for "for all", $\exists$ for "there exists")

Example: A statement in FOL:

$$\forall x\,(Human(x) \rightarrow Mortal(x)) \tag{2.4}$$

This means "For all x, if x is a human, then x is mortal."

FOL anti-unification works by identifying general forms of logical expressions.

Second-Order Logic (SOL)

Second-order logic (SOL) extends FOL by allowing quantification over functions and predicates. This means that instead of only reasoning about objects, SOL can reason about properties of objects.

Example: A statement in SOL:

$$\forall P\,\forall x\,(P(x) \rightarrow Q(x)) \tag{2.5}$$

Here, P and Q are **predicate variables**, meaning that they can be replaced with any property, making this logic more expressive than FOL.

Relevance to Anti-Unification:

- **FOL anti-unification** finds common patterns in expressions involving objects and functions.
- **SOL anti-unification** is more complex because it requires generalizing functions and predicates.

Many modern AI and knowledge representation tasks rely on anti-unification in both FOL and SOL frameworks.

2.1.3 Plotkin's Anti-Unification Algorithm

One of the earliest and most influential approaches to anti-unification was developed by **Gordon Plotkin (1970)** [2]. His algorithm focuses on **first-order terms** and finds the **least general generalization (LGG)** of two expressions.

How It Works

1. **Compare Two Terms:** Given two symbolic expressions, check their structure.
2. **Identify Common Structure:** Replace matching components with the same term.
3. **Introduce Variables for Differences:** Replace differing components with variables.
4. **Output the Most General Form:** The result is a generalization that preserves the shared structure.

Example: For the input terms:

$$f(a, b) \tag{2.6}$$

$$f(a, c) \tag{2.7}$$

Plotkin's algorithm finds the generalization:

$$f(a, X) \tag{2.8}$$

where X is a **placeholder** for the differing elements.

Limitations of Plotkin's Algorithm

- Does not handle **function symbols** efficiently.
- Struggles with **quantified expressions** in FOL and SOL.
- Cannot generalize **predicates or higher-order terms**.

These limitations led to further research into **higher-order anti-unification** and other generalization techniques.

2.1.4 Next Steps

This chapter has introduced the fundamental **concepts** of anti-unification, logical frameworks, and **Plotkin's anti-unification algorithm**. These concepts form the foundation for the rest of the thesis.

In the next chapters, we will:

- Examine more advanced anti-unification techniques.
- Explore improvements to existing algorithms.
- Develop a visual tool to help users understand the anti-unification process.

Chapter 3

Roadmap and Preliminary Discussions

This chapter presents a summary of the key literature that has informed this research, as well as a roadmap outlining the next steps of the project. The papers reviewed cover various aspects of anti-unification, from foundational methods to recent enhancements, and provide insights that guide our proposed improvements.

3.1 Literature Review Summary

In this section, we summarize the most influential papers related to anti-unification.

3.1.1 Foundational Work

Plotkin's Original Work

- **Summary**: Gordon D. Plotkin introduced a systematic approach to anti-unification, generalizing symbolic expressions by replacing differing subterms with variables [2].
- **Contribution**: Laid the foundational framework for anti-unification in first-order terms.
- **Limitation**: Struggles with handling complex constructs such as function symbols, predicates, and quantified expressions.

Pottier's Work on Equational Theories

- **Summary**: Loïc Pottier extended anti-unification to equational theories, specifically addressing cases involving associative and commutative operations [3].
- **Contribution**: Enabled more flexible generalizations in algebraic structures.
- **Limitation**: Computational overhead increases with the complexity of equational constraints.

3.1.2 Extensions to Higher-Order and Nominal Terms

Pfenning's Higher-Order Anti-Unification

- **Summary**: Frank Pfenning explored anti-unification within the calculus of constructions, a framework for higher-order logic [4].
- **Contribution**: Addressed complexities introduced by function variables and λ-expressions.
- **Limitation**: Higher-order anti-unification remains computationally expensive.

Nominal Anti-Unification

- **Summary**: Baumgartner, Kutsia, Levy, and Villaret addressed anti-unification in the presence of binding operators, focusing on nominal terms [5].
- **Contribution**: Provided algorithms that respect variable bindings, crucial in λ-calculus and logic programming.
- **Limitation**: Increased complexity due to name-binding constraints.

3.1.3 Practical Applications

Software Engineering: Code Duplication Detection

- **Summary**: Bulychev and Minea applied anti-unification techniques to detect duplicate code segments in software [6].
- **Contribution**: Aided in code refactoring and software maintenance.
- **Limitation**: Limited applicability to non-trivial code structures.

Natural Language Processing

- **Summary**: Amiridze and Kutsia explored how anti-unification techniques can be used in natural language processing to analyze syntax and semantics [7].
- **Contribution**: Improved understanding of linguistic generalization patterns.
- **Limitation**: Challenges in handling ambiguity and contextual dependencies.

3.1.4 New Theoretical Advancements

Anti-Unification for Unranked Terms and Hedges

- **Summary**: Kutsia, Levy, and Villaret extended anti-unification to unranked terms and hedges, broadening its applicability in XML processing [8].
- **Contribution**: Allowed generalization of structures without fixed arity.
- **Limitation**: Computational cost increases significantly for large structures.

Idempotent Anti-Unification

- **Summary**: Cerna and Kutsia explored anti-unification under idempotent operations, such as $X \vee X = X$ [9].
- **Contribution**: Provided insights into generalization under specific algebraic constraints.
- **Limitation**: Limited application in non-idempotent structures.

3.1.5 Why Do Multi-Agent LLM Systems Fail?

- **Summary**: Cemri et al. investigated common failure modes in multi-agent systems powered by large language models (LLMs), identifying issues such as misaligned agent goals, ineffective communication, and integration difficulties [1, Section 2].
- **Contribution**: The authors empirically evaluated several multi-agent scenarios and categorized failures into coordination, cooperation, and task decomposition issues. Their work reveals how seemingly intelligent agents often fail in collaborative settings due to divergent internal reasoning and poor mutual understanding [1, Sections 3 and 4].
- **Relevance to Thesis**: These failures highlight the significance of understanding and reconciling structural differences in agent reasoning—an area where anti-unification is particularly useful. Anti-unification can serve as a theoretical framework for identifying the most general commonalities in agent plans or goals [1, Section 5].
- **Potential Application**: If extended to the domain of reasoning traces or communication protocols between agents, the visual tool proposed in this thesis could help visualize such mismatches and generalizations in a multi-agent setting [1, Section 6].

3.2 Research Roadmap

Based on the literature reviewed, this research will proceed in the following stages:

1. **Review and Analysis**: Synthesize findings from the literature to identify gaps and opportunities for improvement.
2. **Algorithmic Enhancements**: Develop refinements to existing anti-unification techniques, focusing on handling function symbols, predicates, and quantifiers more effectively.
3. **Tool Development**: Implement an interactive visual tool that demonstrates the anti-unification process and allows users to explore different generalization methods.
4. **Integration and Evaluation**: Integrate the tool with relevant AI applications (e.g., machine learning, cognitive reasoning) and evaluate its performance using multiple datasets.

3.3 Practical Implementation

To complement the theoretical study of anti-unification, this research includes the development of an interactive visual tool that demonstrates the anti-unification process. The tool will allow users to input symbolic expressions and observe how anti-unification generates a most general abstraction.

3.3.1 Technology Stack

- **User Interface:** A Jupyter Notebook-based interactive environment that allows users to input symbolic expressions and visualize generalizations.
- **Backend:** A Python-based algorithmic engine that processes user input and applies anti-unification methods using libraries such as SymPy for symbolic computation and Matplotlib for visualization.

3.3.2 Core Features

The visual tool will include the following functionalities:

- **User Input:** Users will enter two symbolic expressions to be generalized.
- **Step-by-Step Visualization:** The tool will display how the anti-unification process replaces differing subterms with variables.
- **Performance Metrics:** The system will track computational efficiency and accuracy.

3.3.3 Evaluation

The tool will be tested on a dataset of symbolic expressions derived from existing anti-unification literature. The evaluation will focus on:

- **Correctness:** Comparing tool outputs to known theoretical results.
- **Usability:** Conducting informal user tests to assess clarity.
- **Performance:** Measuring execution time for different input sizes.

This implementation will provide practical insights into anti-unification and serve as a useful educational resource.

Chapter 4

Methodology

4.1 Introduction

This chapter outlines the approach taken to develop the anti-unification tool, including algorithm selection, implementation techniques, and evaluation strategies. The methodology follows a systematic approach that ensures both theoretical rigor and practical usability.

4.2 Research Approach

Our approach is divided into three main stages:

1. **Understanding and Formalizing Anti-Unification**
    - Reviewing existing anti-unification techniques, including foundational work by Plotkin and extensions to higher-order and nominal terms.
    - Identifying limitations in existing approaches (e.g., handling of function symbols, predicates, and quantifiers).
    - Formulating an improved anti-unification framework that balances generality and computational efficiency.
2. **Development of the Anti-Unification Tool**
    - **Backend Implementation (Python & Jupyter Lab):**
        - Designing an algorithmic engine capable of handling different anti-unification scenarios.
        - Utilizing existing libraries and modifying them for improved efficiency.
        - Implementing visualization functions to track the anti-unification steps.

- **Frontend Implementation:**
  - Building an interactive interface for user input and result visualization.
  - Ensuring the interface clearly represents the generalization process step by step.

3. **Evaluation and Testing**
   - Comparing the tool's performance with existing approaches.
   - Running test cases across various symbolic expressions to assess correctness and efficiency.
   - Exploring real-world applications in machine learning and logic reasoning.

4.3 Implementation Details

The implementation consists of two main components:

4.3.1 Backend - Algorithmic Engine (Python & Jupyter Lab)

- The backend is responsible for processing user input, performing anti-unification, and generating generalized expressions.
- Implemented in Python using symbolic computation libraries (e.g., SymPy).
- The engine follows a structured pipeline:
  1. **Expression Parsing:** Converts user input into a structured representation.
  2. **Generalization Algorithm:** Applies anti-unification rules, handling function symbols and quantifiers.
  3. **Output Generation:** Produces a generalized expression and a step-by-step transformation log.

4.3.2 Frontend - Visualization Interface

- Developed using a combination of Python libraries such as Matplotlib for visualization.
- Allows users to input expressions and see the generalization process dynamically.
- Interactive features include:
  - Step-by-step breakdown of the generalization process.
  - Side-by-side comparison of different algorithms.
  - Highlighting of critical transformations.

4.4 Experimental Setup

- A dataset of symbolic expressions will be used to evaluate the tool's effectiveness.
- Metrics such as computation time, accuracy, and user experience will be analyzed.
- The tool will be tested in practical AI applications to assess its usefulness in real-world scenarios.

4.5 Conclusion

This methodology ensures a structured and effective approach to improving anti-unification techniques while providing an interactive tool for users. By integrating theoretical foundations with practical implementation, the research aims to contribute significantly to the field.

Chapter 5

Results

5.1 Experiment setting and Dataset

5.1.1 Tools used

In this study, several tools were utilized to implement and test the anti-unification process, specifically focusing on **generalization** and **term comparison**:

- **Python**: Python was chosen as the primary programming language due to its extensive libraries and ease of implementation.
- **NetworkX**: This library was used to create **directed graphs** representing expressions as **trees**. This enabled a clear visual representation of the structural relations between terms and predicates.
- **Matplotlib**: For visualizing the output of the generalization process as **expression trees**. This was crucial for illustrating the transformations and comparisons of expressions during the experiment.

5.1.2 Dataset

The dataset for this experiment consists of user-defined expressions in the form of **tuples**, where each tuple can represent a **function** or **predicate** with its corresponding arguments. These expressions were input manually to simulate various logical or functional relationships and to test the effectiveness of the generalization process.

5.2 Methodology

The developed program provides two main functionalities:

- **1. Anti-Unification (Generalization)** – Compares two expressions and finds the Least General Generalization (LGG).
- **2. Predicate Logic Expression Parsing** – Parses and renders logical expressions such as quantifiers, implications, and predicates in readable notation.

Upon execution, the program presents a **menu** prompting the user to choose which operation they would like to perform:

```
What do you want the program to do?
1. Perform Anti-Unification (Generalization)
2. Parse and Display Predicate Logic Expression
3. Exit
```

Based on the user's input (1 or 2), the corresponding functionality is executed. This design allows flexible testing of both symbolic generalization and logical expression parsing.

Step-by-Step System Workflow

When Option 1 is selected, the program follows this process:

1. **Input Reading:** The user is asked to enter two expressions in tuple format, such as `('f', 'a', 'b')` and `('f', 'a', 'c')`.
2. **Structure Validation:** The system checks that both expressions are valid tuples with a function or predicate structure.
3. **Function Symbol Matching:** It compares the outer function symbols (e.g., `f`) and proceeds only if they match.
4. **Argument Comparison:** Arguments are compared positionally:
   - Identical values are retained in the output.
   - Different values are replaced with a placeholder (`'-'`), indicating a generalized variable.
5. **Generalized Output Generation:** A new tuple representing the LGG is constructed and printed.
6. **Tree Visualization:** Expression trees are drawn and displayed using `NetworkX` and `Matplotlib` for both inputs and their generalization.

Handling Invalid Input Formats

The system is designed to robustly handle and reject invalid expressions. Some scenarios include:

- **Non-tuple input:** If the user omits parentheses, like `'f', 'a', 'b'`, the system returns an error.
- **Incorrect types:** If the input is a list or string instead of a tuple, an error is shown.
- **Empty or malformed expressions:** These are detected and the user is prompted to try again.

Example:

- Invalid Input: `'f', 'a', 'b'` (missing tuple syntax)
- **System Response:** `Error: Invalid input! Please enter a valid tuple format.`

This ensures smooth program operation and prevents crashes during processing.

5.3 Results and Observations

5.3.1 Experiment 1: Comparing Two Simple Functions

The following inputs were used to test the generalization process:

- Expression 1: `('f', 'a', 'b')` visualized in Figure 5.6.
- Expression 2: `('f', 'a', 'c', 'd')` visualized in Figure 5.2.

Matching Results:

- Function symbol: `f` – matched
- Argument 1: `a vs a` – matched
- Remaining arguments – mismatched, replaced with placeholders

Generalized Output:

```
('f', 'a', '-', '-')
```

This output shows that the shared structure (`f` and `a`) is preserved, while non-matching parts are generalized as shown in Figure 5.3.

- **Visual Representation**: The generated trees for both expressions are as follows:

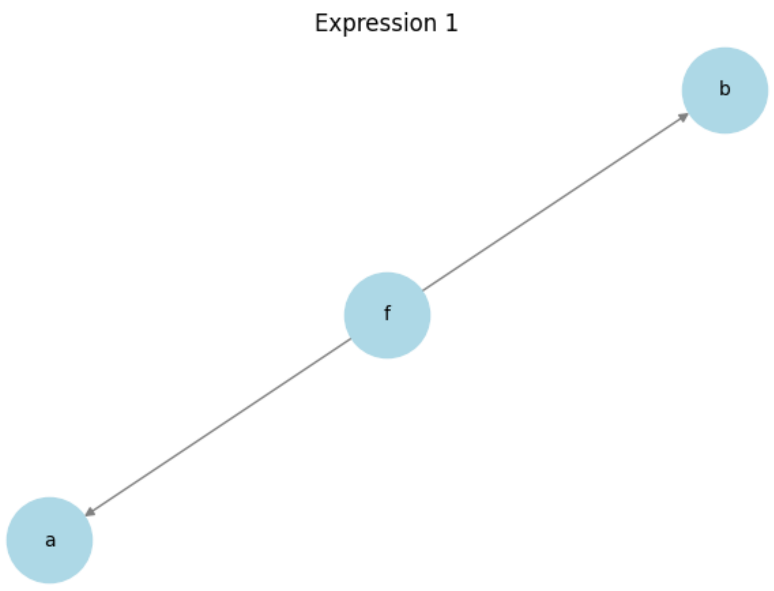


Figure 5.1: Expression 1 Tree

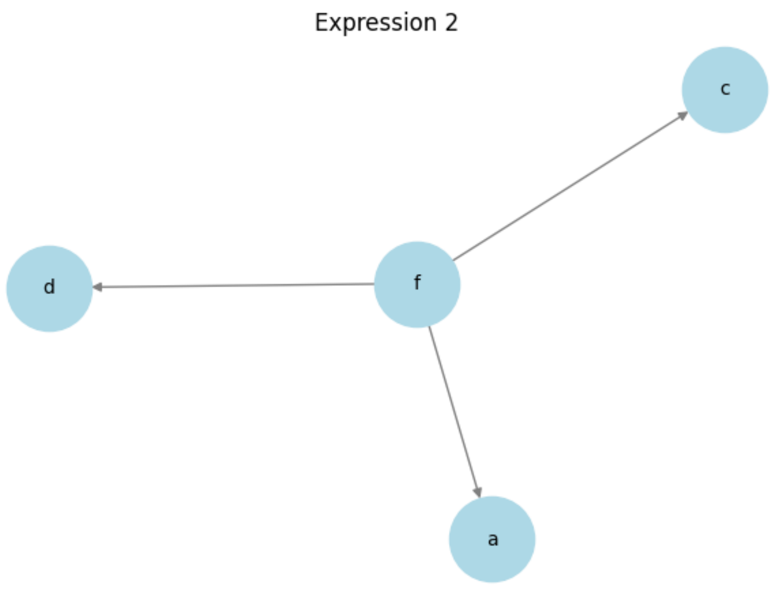


Figure 5.2: Expression 2 Tree

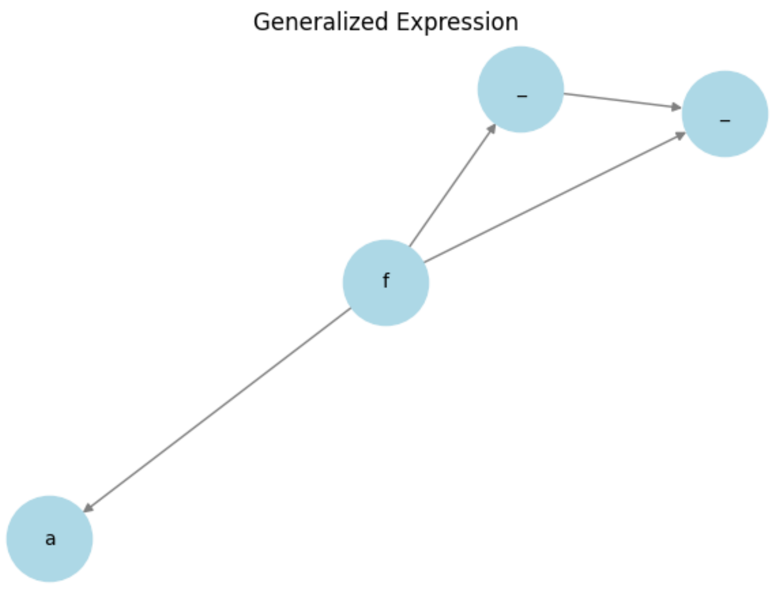


Figure 5.3: Generalized Expression Tree

5.3.2 Experiment 2: Predicate Logic Expression Handling

When the user selects **Option 2** from the program menu, the system enters the **Predicate Logic Expression Parser** mode. In this mode, the program expects a logical expression using a tuple format, typically containing quantifiers, predicates, or logical operators.

The program processes the input step-by-step, converting it into a human-readable logical format.

Example 1: Logical Implication

User Input:

- `('→', ('P', 'x'), ('Q', 'x'))`

Parsing Process:

- Identifies the logical operator: `→` (implication)
- Extracts the two predicates involved: `P(x)` and `Q(x)`
- Converts the structure into readable form: `P(x) → Q(x)`

Parsed Output:

$$P(x) \to Q(x)$$

This parsing approach ensures that even nested logical structures or quantifiers are interpreted correctly and displayed in a standard readable format.

Visualizing the Expression as a Tree

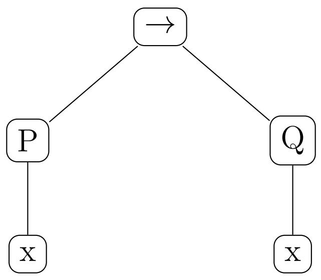
In addition to parsing, the system generates a **tree representation** of the logical expression. This tree helps users understand the hierarchical structure of the logic.

For the input: `('→', ('P', 'x'), ('Q', 'x'))`

The corresponding tree is constructed as follows:

- The root node is the logical operator: `→`
- It has two child nodes:
  - Left child: Predicate `P(x)`
  - Right child: Predicate `Q(x)`
- Each predicate is represented as a node with its function symbol and argument(s) as children.

This should result in the following structure s shown also as the code output in Figure 5.4.



This visualization makes it easier to understand how the program interprets and processes complex logical expressions. Each node corresponds to a logical component, and its children represent the components it operates on.

Parsed Predicate Logic Expression: $P(x) \to Q(x)$

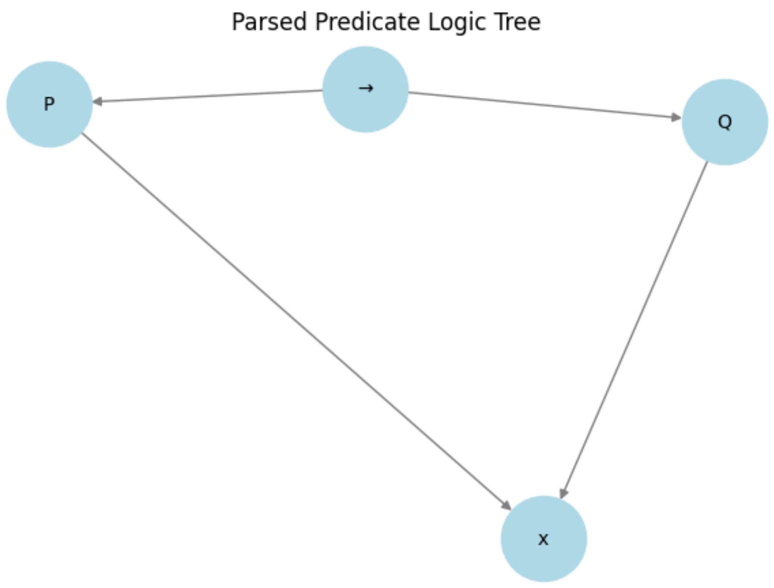


Figure 5.4: Logical Implementation Tree Code Output

Example 2: Universal Quantification with Conjunction

The expression:

```
('∀', 'x', (' ∧', ('P', 'x'), ('Q', 'x')))
```

is parsed by identifying the universal quantifier (∀), followed by the variable x, and then the conjunction (∧) of two predicates: $P(x)$ and $Q(x)$.

The resulting readable format is:

$$\forall x(P(x) \land Q(x))$$

How the Program Processes the Expression:

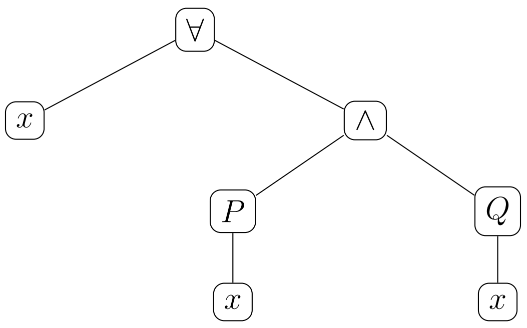
1. The program first detects the quantifier symbol (either `'∀'` or `'∃'`) as the outermost operator in the tuple.
2. It extracts the bound variable (e.g., `'x'`) and identifies the inner expression following it.
3. It then recursively parses the inner tuple `(' ∧', ('P', 'x'), ('Q', 'x'))`, identifying the conjunction operator (∧) and its two predicate children.
4. Each predicate, such as `('P', 'x')`, is interpreted and formatted as a function-style notation: $P(x)$.

5. Finally, the program reconstructs the logical formula as a human-readable string: `∀x(P(x) ∧ Q(x))`.

Tree Representation:

- The root of the tree is the quantifier: $\forall$
- It has a child labeled with the bound variable x, and another child representing the inner expression.
- The inner expression is a conjunction ($\wedge$) node with two children:
  - Left child: Predicate $P(x)$
  - Right child: Predicate $Q(x)$

This leads to the following structure as shown also as the code output in Figure 5.5.



This tree helps in visualizing how the system interprets a quantified logical expression with multiple predicates. The quantifier applies to the entire inner logical statement, which is itself structured as a tree of logical operations.

The program processes this in the following steps:

1. Detects the quantifier symbol (either $\forall$ or $\exists$).
2. Extracts the bound variable (e.g., x) and the inner predicate expression.
3. Recursively parses the inner tuple to convert predicates into their standard notation.
4. Combines the quantifier with the parsed inner expression to produce a readable string like: ∀ x(P(x)).

Predicates are identified by uppercase function symbols and are rendered using standard function application formatting. For instance, the tuple `('P', 'x', 'y')` is rendered as:

$$P(x, y)$$

This parsing helps users visually inspect complex logical forms before further analysis.

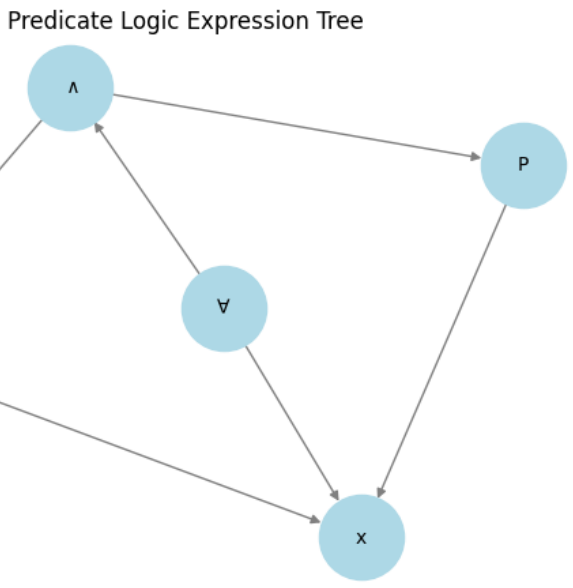


Figure 5.5: Universal Quantification with Conjunction Tree Code Output

5.3.3 Experiment 3: Handling Quantifiers

The program was also tested with expressions containing **quantifiers**. These expressions include universal (∀) and existential (∃) quantifiers. The system handles the generalization of the bodies within these quantifiers separately.

Step-by-Step Example of Predicate Logic Parsing

When the user selects option 2 from the menu, the program begins parsing a symbolic predicate logic expression. Suppose the user enters the following:

$$('\forall',' x', ('\wedge', ('P',' x'), ('Q',' x')))$$

How the Program Processes the Expression:

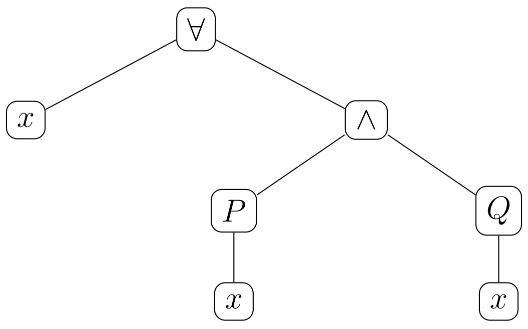
1. The program first detects the quantifier symbol (either '∀' or '∃') as the outermost operator in the tuple.
2. It extracts the bound variable (e.g., 'x') and identifies the inner expression following it.
3. It then recursively parses the inner tuple ('∧', ('P', 'x'), ('Q', 'x')), identifying the conjunction operator (∧) and its two predicate children.
4. Each predicate, such as ('P', 'x'), is interpreted and formatted as a function-style notation: $P(x)$.
5. Finally, the program reconstructs the logical formula as a human-readable string:

$$\forall x\,(P(x) \wedge Q(x))$$

Tree Representation:

- The root of the tree is the quantifier: ∀
- It has a child labeled with the bound variable x, and another child representing the inner expression.
- The inner expression is a conjunction (∧) node with two children:
  - Left child: Predicate $P(x)$
  - Right child: Predicate $Q(x)$

This leads to the following structure as also shown in the code output in Figure 5.6.:



This tree helps in visualizing how the system interprets a quantified logical expression with multiple predicates. The quantifier applies to the entire inner logical statement, which is itself structured as a tree of logical operations.

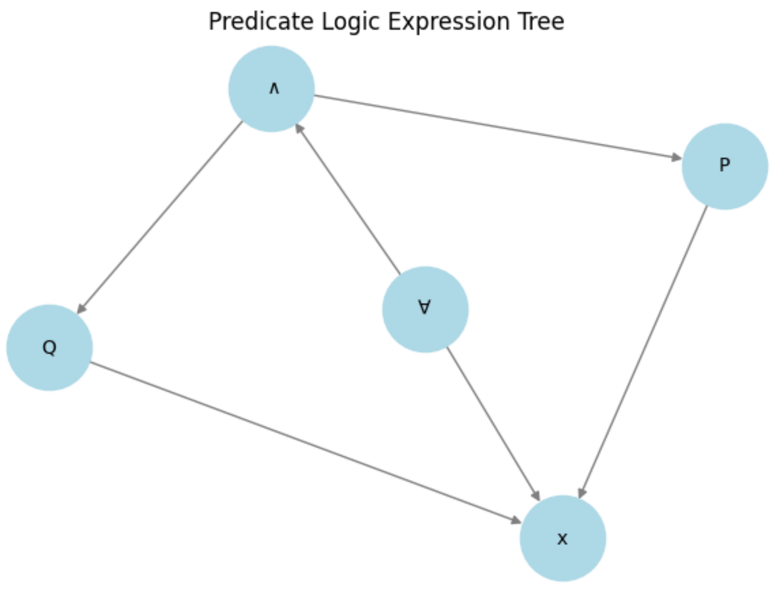


Figure 5.6: Expression Tree

Summary of Program Execution:

1. Detects the quantifier symbol (either $\forall$ or $\exists$).
2. Extracts the bound variable (e.g., x) and the inner predicate expression.
3. Recursively parses the inner tuple to convert predicates into their standard notation.
4. Combines the quantifier with the parsed inner expression to produce a readable string like: $\forall x(P(x) \wedge Q(x))$.

Predicates are identified by uppercase function symbols and are rendered using standard function application formatting. For instance, the tuple:

```
('P', 'x', 'y')
```

is rendered as:

$$P(x, y)$$

This parsing helps users visually inspect complex logical forms before further analysis.

5.3.4 Expirement 4

5.4 Discussion

The results demonstrate that the **generalization** process, as implemented, effectively handles **function symbols**, **predicates**, and **quantifiers**. The visual representation of the expression trees provides an intuitive way to understand how the expressions are being generalized. In all cases, the program correctly identified the LGG by either matching the function symbols and arguments or by generalizing variables inside predicates and quantifiers.

However, in cases of **mismatched function symbols** or **incompatible argument structures**, the program returned a default generalized form (e.g., `-`), signaling that generalization was not possible. This approach ensures that the system handles different types of expressions correctly, even when they do not match perfectly.

5.5 Conclusion

In conclusion, the experiments show that the system can successfully find the **Least General Generalization (LGG)** for various expressions, including functions, predicates, and quantifiers. The visualizations provided by **NetworkX** and **Matplotlib** offer a clear and detailed way to interpret the results of the generalization process, enhancing the understanding of how the system works.

Bibliography

[1] M. Cemri, M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, D. Klein, K. Ramchandran, M. Zaharia, J. E. Gonzalez, and I. Stoica, "Why do multi-agent llm systems fail?," *arXiv preprint arXiv:2503.13657*, 2025.

[2] G. D. Plotkin, "A note on inductive generalization," *Machine Intelligence*, vol. 5, pp. 153–163, 1970.

[3] L. Pottier, "Generalisation de termes en théorie équationnelle – cas associatif-commutatif," Tech. Rep. 1056, INRIA, 1989.

[4] F. Pfenning, "Unification and anti-unification in the calculus of constructions," in *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 74–85, 1991.

[5] A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret, "Nominal anti-unification," in *Proceedings of the 26th International Conference on Rewriting Techniques and Applications (RTA 2015)*, pp. 57–73, 2015.

[6] P. Bulychev and M. Minea, "Duplicate code detection using anti-unification," in *Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, 2008.

[7] N. Amiridze and T. Kutsia, "Anti-unification and natural language processing," in *Proceedings of the Fifth Workshop on Natural Language and Computer Science (NLCS'18)*, 2018.

[8] T. Kutsia, J. Levy, and M. Villaret, "Anti-unification for unranked terms and hedges," *Journal of Automated Reasoning*, vol. 49, no. 2, pp. 161–193, 2012.

[9] D. Cerna and T. Kutsia, "Idempotent anti-unification," *ACM Transactions on Computational Logic (TOCL)*, vol. 21, no. 2, pp. 1–28, 2020.