Bachelor's thesis

On the Usability of the Guarded Family of Rule-based Ontology Languages

Ioannis Michail

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2026

“Δηλώνω ότι η παρούσα διατριβή και οποιοδήποτε συνοδευτικό λογισμικό αποτελούν δικό μου πρωτότυπο έργο, εκπονημένο σε πλήρη συμμόρφωση με τον Κώδικα Δεοντολογίας για την Έρευνα του Πανεπιστημίου Κύπρου (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), και με πλήρη ευθύνη εκ μέρους μου για την ακρίβεια, την ακεραιότητα και την εγκυρότητα όλου του περιεχομένου.
I declare that this dissertation and any accompanying software are our own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on our part for the accuracy, integrity, and validity of all content.”

AI was used to assist with the structure of the document. Also, it was used as a language assistant translating small sentences from Greek to English and polishing the writing by correcting grammatical and spelling issues since English is not the author’s native language

Abstract

This thesis studies the Guarded Family of Tuple-Generating Dependencies languages and experimentally investigates their occurrence in real datasets. These languages are restricted forms of logical rules that aim to achieve a balance between expressiveness and computational efficiency, while limiting the uncontrolled propagation of null values.

In our work, a rule analysis system was developed in Java, which automatically recognizes whether a ruleset belongs to each of the 6 languages our study is based on. These languages are Linear, Guarded, Frontier-Guarded, Weakly-Guarded, Weakly-Frontier-Guarded and Warded. The system includes mechanisms for positions analysis, variables categorization and checker for each language's theoretical properties. In addition, it includes a preprocessing script and a script used for automation procedure of the experiment implemented in Python.

The experimental evaluation was carried out on more than 18500 real datasets. The results showed that all Guarded family languages appear at high percentages in real data which proves their theoretical constrains reflect the structure of real datasets. One of the main findings of this work concerns the Warded language, which covers approximately 85% of the datasets, while maintaining better computational properties compared to other expressive languages of this rate.

Overall, the work contributes both theoretically and practically, as it transforms the theoretical definitions of all Guarded Family languages into implementable algorithms and provides experimental evaluation and analysis on real data.

Table of Contents

Chapter 1 Introduction *1*

Chapter 2 Preliminaries *7*

2.1 Introduction **7**

2.1.1 Atom 8

2.1.2 Structure of a Rule 8

2.1.3 Tuple-Generating Dependencies 9

2.1.4 Existential Variables **10**

2.2 Variables & Positions **10**

2.2.1 Positions 10

2.2.3 Affected vs Non-affected Positions 12

2.2.5 Datasets and language properties 16

2.3 Linear TGDs **17**

2.4 Guarded TGDs **18**

2.5 Frontier-Guarded TGDs **20**

2.6 Weakly-Guarded TGDs **22**

2.7 Weakly-Frontier-Guarded TGDs **26**

2.8 Warded TGDs **30**

Chapter 3 Implementation Details *35*

3.1 System Overview / Architecture **36**

3.2 Preprocessing **38**

3.3 Rule Analysis and System Representation **38**

3.3.1 Parsing of the rules 38

3.3.2 Internal Representation of Rules 39

3.3.3 Representation of positions and their importance 40

3.3.4 Auxiliary variable analysis 41

3.3.5 Categorization of variables and GlobalWardedAnalysis 41

3.3.6 Role of the App.java and App2.java files 42

3.3.7 Overall significance of the stage 47

3.4 Classification Engine **48**

3.4.1 Overall operation of the mechanism 49
3.4.2 Linear Checker 49
3.4.3 Guarded Checker 50
3.4.4 Frontier-Guarded Checker 50
3.4.5 Weakly-Guarded Checker 51
3.4.6 Weakly-Frontier-Guarded Checker 51
3.4.7 Warded Check 52

3.5 Challenges and Implementation Difficulties 53

3.5.1 Understanding and converting theoretical concepts 54
3.5.2 From local analysis to global ruleset analysis 54
3.5.3 Dataset Inconsistencies and preprocessing issues 55
3.5.4 implementation and consistency of data structures 55
3.5.5 Performance and efficiency 56
3.5.6 Validation and debugging 56
3.5.7 Overall assessment 57

3.6 Implementation Summary and Transition to Experiments 57

Chapter 4 Experimental Evaluation 59

4.1 Datasets 60

4.2 Data Preprocessing 60

4.3 Experimental Setup and Reproducibility 63

4.4 Execution Pipeline (batch_analyze.py) 68

4.5 Experimental Results 73

4.5.1 Total number of datasets 73
4.5.2 Distribution of datasets by language 74
4.5.3 Interpretation of results 74
4.5.4 Expressiveness comparison 76
4.5.5 Balance of expressiveness and complexity 77
4.5.6 Observation for real datasets 77
4.5.7 Visualization of results 78
4.5.8 Analysis based on dataset size 83

4.6 Discussion and Interpretation of Results 87

4.6.1 Confirmation of the theoretical hierarchy 87
4.6.2 Behavior of real datasets 88

4.6.3 The role of null propagation 88
4.6.4 Syntactic vs Semantic restrictions 89
4.6.5 Limitations of aggregate analysis 89
4.6.6 Practical implications 89

Chapter 5 Conclusions 91

Chapter 1

Introduction

In recent years, the need to represent, manage and process knowledge has gained importance in the fields of databases, artificial intelligence and knowledge graphs. Modern information systems are required to represent complex relationship between entities, extract knowledge from data and support logical reasoning rather than simply storing data. This led to the development of rule-based knowledge representation languages, which allow the production of new information through logical rules while restricting the null values.

One of the most important models in this field is Tuple-Generating Dependencies (TGDs), which are logical rules that describe how new data can be generated throughout some requirements. Each rule has a body and a head, the body represents the conditions that must be met while the head represents the new information that can be extracted. TGDs are widely used in applications that have to do with data integration, ontology-based data systems and generally in logical reasoning systems.

However, the expressiveness of TGDs comes with a significant computational cost. Some rules may be capable to introduce new null values that represent unknown objects whose existence in implied by the rule. This can lead to serious computational problems as null values may generate even more nulls and the continuous creation of null values becomes infinite or computationally very expensive.

This created a fundamental problem, expressive logical rules are needed in order to model real world knowledge in a powerful way but at the same time, unrestricted use of such rules may make basic reasoning tasks very computational. Therefore, the last few years a major research direction in database theory and knowledge representation has been the creation of restricted classes (languages) of TGDs that keep expressiveness while providing better computational behavior without null propagation.

These restricted languages aim to strike a balance between expressiveness and efficiency. A language that is too restrictive may have good computational properties but may not be really useful to real world problems if it does not accept a large percentage of real-world rules. However, a language that is too expressive may include most of the rules, but at the same time it might be very difficult to use efficiently in practice. This implies that the most interesting and practically useful languages are those that strike a balance between these two requirements.

One important family of such languages is the Guarded Family of TGDs which is also the family this study is based on. This family includes well-known classes of rules such as Linear TGDs, Guarded TGDs, Frontier-Guarded TGDs, Weakly-Guarded TGDs, Weakly-Frontier-Guarded TGDs and Warded TGDs. All these languages impose a different structural restriction on the rules and have different results at expressiveness and efficiency. Their common objective is to control the way variables may carry null values.

The simplest and most restricted language in our study is Linear TGDs. It has a very simple structure of the rules it accepts and the possible data combinations are limited, it doesn't accept joins between multiple body atoms so it's not so practical.

Guarded TGDs are a more expressive extension of Linear TGDs. They introduce the term guard atom which is which is also used by other languages with different criteria for what this atom should protect, this atom ensures that body variables are connected which limits uncontrolled data combination. Despite these Guarded TGDs are still a restrictive language since they have a restrictive rule for the guard atom.

Frontier-Guarded TGDs relax this restriction further. They require from the guard atom to restrict less variables and more specifically those that appear both in the body and the head, so it's a more expressive language but it's still one of the strictest languages.

Weakly-Guarded TGDs is the first language on our study that belongs to the expressive languages as it introduces a new idea that not all variables all equally dangerous, some variables appear in positions that are dangerous and some others appear only in safe positions, variables have categories based on positions they appear that are harmless,

harmful and dangerous and Weakly-Guarded focuses on harmful and dangerous variables requiring them in a common body atom.

Weakly-Frontier-Guarded TGDs extend this idea further focusing exclusively on dangerous variables and not just harmful as they are considered the most critical variables and they transfer null values from the body to the head. This is the most expressive language in the Guarded Family but also the most computationally expensive.

Warded TGDs are another language of the Guarded Family that is based on the classification of variables, and especially on dangerous variables. In this language, dangerous variables must occur together in a specific body atom called the ward. In addition, the ward is allowed to share with the rest of the body only harmless variables. Therefore, Warded TGDs control the propagation of null values through a structural restriction on dangerous variables, in the same way that the other Guarded Family languages impose their own restrictions on the form of rules.

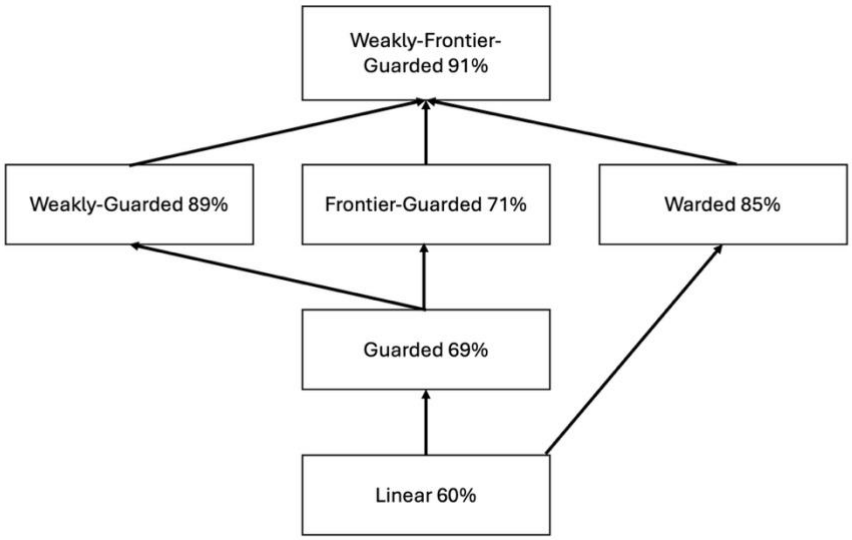
Although the theoretical properties of Guarded Family languages have been studied, an important research question remains open: To what extend do these languages actually appear in real datasets and ontologies? So, the problem is if the structural constrains proposed in theory actually reflect to the structure of real datasets and can be applied to them effectively or if it cannot be implemented effectively.

This is the main motivation of the present thesis. Our work is not only limited to the theoretical study of Guarded family languages, but our goal is to experimentally investigate their practical presence in real datasets. This approach attempts to relate theory and practice, and we present an evaluation of which languages appear most often and which languages provide balanced and realistic model for real knowledge systems.

It is important to clarify that all languages in the Guarded Family are examined in this thesis in the same objective way. The purpose of the work is not to promote one language over the others, but to compare their practical presence in real datasets using the same methodology and the same classification criteria for every language. Each language is presented according to its theoretical definition, implemented through its corresponding

checker, and evaluated on the same collection of datasets. Therefore, any additional emphasis given to a specific language in the later analysis is a consequence of the experimental results and not an initial assumption of the study.

Now that we have talked about the language families and the purpose of our work, I will present the Guarded Family Tree that shows what relationships exist between languages, which language is a subset of which, and the results. An explanation of the results and how we arrived at them will follow in the sequel.



On the graph it shows that the basis is Linear TGDs, if a rule is Linear then it automatically belongs to all the other languages in the Guarded Family. Next, we have Guarded which is the basis for Frontier-Guarded, Weakly-Guarded and Weakly-Frontier-Guarded, then we see Frontier-Guarded, Weakly-Guarded and Warded have no direct relationship between them but constitute the basis of Weakly-Frontier-Guarded which is the most expressive language of all.

As for the results, there is a detailed explanation in Chapter 4, however we can already discuss the fact that all languages have relatively high rates, none can be characterized as non-applicable. It seems that the higher we go in the tree, the higher the rates are and the

languages that have high rates and stand out are the Warded, Weakly-Guarded and Weakly-Frontier-Guarded. These results are the answer to the question "To what extend do these languages actually appear in real datasets and ontologies?".

In this work, a comprehensive ruleset analysis framework was developed, which can identify whether a ruleset belongs to the Linear, Guarded, Frontier-Guarded, Weakly-Guarded Weakly-Frontier-Guarded and Warded languages. A ruleset is considered to belong to a language only if all of its rules belong to that language. The implementation was created in Java and includes several individual subsystems. Each rule is represented thought its atoms and variables and there are analysis mechanisms for positions, position categorization, variable categorization and everything needed for language checkers.

For every language there is a checker class, for simpler languages the checking in mainly based on the structure of the rule while for more expressive languages the check is based on variables classification and more complex analysis. For this reason, the system performs a global analysis of the entire ruleset, calculating all positions and classifying all variables and only after that the checkers have what they need to see if each rule meets the requirements of each language.

In addition to the implementation, the work also includes a ruleset preprocessing pipeline. The datasets were not in a format acceptable by our system so a Python script was developed that cleans and normalizes the rules so they can be analyzed safely.

Also, a second Python script was created to automate the experiments. The scripts automatically run the analyzer on rulesets, creating result files and statistics for the datasets. This automation was necessary due to the large number of datasets analyzed.

The experimental part of the work involves the analysis of more than 18000 datasets. The system checks for each dataset in how many of the 6 languages we study it belongs and then calculate the total percentages for each language.

The results of the work lead to two particularly important conclusions. The first main finding is that all languages of the Guarded Family show relatively high rates of presence in real datasets. This fact suggests that the theoretical constraints proposed in the literature are not just abstract mathematical constructs but reflect real patterns that often occur in real rulesets. The second and most important finding concerns the Warded language. The experimental results show that approximately 85% of the datasets analyzed belong to the Warded language. This is an important result as it shows that Warded covers a lot of datasets while being a very efficient language, Weakly-Guarded and Weakly-Frontier-Guarded cover more datasets but they are very computational and Linear, Guarded and Frontier-Guarded are efficient but they do not cover a very satisfactory number of datasets. Warded is between these two categories and manages to achieve an excellent balance between expressiveness and efficiency as it is expressive enough to cover most datasets and have efficient computational properties and it is the most practically balanced language to use for modeling real world datasets.

Overall, this paper contributes both theoretically and practically. At theoretical level, it presents in an analytical manner with explanation the languages of the Guarded Family, the relationships between them and role of positions and variables. At practical level it translates the theoretical definitions into implementable algorithms and provides extensive experimental evaluation on real datasets.

The rest of the paper is organized as follows:
-Chapter 2 presents the theoretical background with all the concepts related to TGDs analyzed and the Guarded Family.
-Chapter 3 describes the implementation of the analyzer and the algorithms developed explaining what each class does and how they are connected
-Chapter 4 presents the experimental evaluation with all the steps needed to produce the result and further analysis of the results
-Chapter 5 presents the conclusions of the paper and possible directions for future research

Chapter 2

Preliminaries

2.1 Introduction **7**
2.1.1 Atom 8
2.1.2 Structure of a Rule 8
2.1.3 Tuple-Generating Dependencies 9
2.1.4 Existential Variables **10**
2.2 Variables & Positions **10**
2.2.1 Positions 10
2.2.3 Affected vs Non-affected Positions 12
2.2.5 Datasets and language properties 16
2.3 Linear TGDs **17**
2.4 Guarded TGDs **18**
2.5 Frontier-Guarded TGDs **20**
2.6 Weakly-Guarded TGDs **22**
2.7 Weakly-Frontier-Guarded TGDs **26**
2.8 Warded TGDs **30**

2.1 Introduction

Propositional logic is a powerful tool to represent knowledge and extract information from databases. However, the full expressive power of these logics often leads to computational problems such as non-terminality of algorithms and high complexity in query evaluations.

In rule-based database environments like the ones we are studying, some variables lead to the creation of null values, which can spread very quickly during the process. This

results in infinite data generation and makes the database not able to efficiently answer queries.

To address these problems, there has been extensive research on constrained languages and it is an active research direction the last few years. These languages impose structural constraints to the rules with the goal to control the null values and to ensure termination and efficient query evaluation.

Overall, these constrained language families achieve compromise between expressiveness and computational efficiency, making them eligible to implement them in databases but with the cost of missing out on some rules.

To understand the Guarded Family and their definitions it is necessary to first present some basic concepts.

2.1.1 Atom

Logical rules in Datalog are constructed using atoms, they are the basic building blocks for rules.

Definition:
An atom is an expression of the form
R(t1,...,tn)
Where R is an n-ary predicate where t1,…,tn are variables.

We write var(α) for the set of variables in an atom α, this notation extends to sets of atoms. [2]

2.1.2 Structure of a Rule

A rule of the form body → head has 2 parts:
The body is a set of atoms that show the conditions of the rule. The head describes new information that can be extracted if the body is satisfied. It also includes atoms and sometimes existential variables (explained in Chapter 2.1.4).

Also, there are variables that appear only in the body and variables that appear only in the head, there are no restrictions about that. It is very important to understand the structure of a rule as there are rule-based languages that are defined based on the structure.

For example:

Parent(x) → Ancestor(y)
Body: Parent(x)
Head: Ancestor(y)

Ancestor(x,z) :- Parent(x,y) & Brother(y,z)
Body: Parent(x,y) & Brother(y,z)
Head: Ancestor(x,z)

We write body(σ) for the set of atoms in the body of a rule σ.
We write head(σ) for the set of atoms in the head of a rule σ.
We write var (body(σ)) for the set of variables in the body of a rule σ
We write var (head(σ)) for the set of variables in the head of a rule σ

2.1.3 Tuple-Generating Dependencies

First of all, TGDs (Tuple-Generating Dependencies) are a type of logical rules used to extract information from databases as we mentioned earlier. They are the basic mechanism which more restricted languages are defined by adding contains to the structure of TGDs.

A TGD has the general form:
body → head
Also, another popular way of expressing TGDs rules is:
head :- body

It is exactly the same and does not change the meaning of the logical proposition, the difference is only syntactic.

2.1.4 Existential Variables

In a TGD rule, the variables that appear in the body are universally quantified and represent the values that must satisfy the body of the rule. Some variables may also appear in the head without appearing in the body. These variables are existentially quantified and are called existential variables. They do not correspond to already known values from the body; instead, they state that there must exist some value that satisfies the head. This is different from frontier variables, which also appear in the head but are not existential because they already appear in the body and therefore transfer existing values from the body to the head. During rule application, existential variables may introduce fresh null values into the database, which makes them important but also potentially dangerous for reasoning.

For example:
Person(x) → ∃y HasChild(x,y)
∃ represents existential quantification
y is an existential variable

2.2 Variables & Positions

2.2.1 Positions

In the context of TGDs, a position refers to a specific position within a predicate (an atom). It is defined as:
predicate + index
If a predicate has n arguments, then his positions are:
R[1], R[2], … , R[n]
For example:
A(x) & R(x,y,z) → C(y,z)

The positions are:
A[1], R[1], R[2],R[3],C[1],C[2]

The variables assigned in each position are:
A[1]=x
R[1]=x
R[2]=y
R[3]=z
C[1]=y
C[2]=z

Positions allow us to refer precisely where variables appear within an atom and in a ruleset This is very important because we are interested in which position variables appear in order to draw conclusions from TGDs rules and not only which variables are present.

2.2.2 Frontier Variables

Frontier Variables are variables that appear both in the body and the head of a rule.
Typically, they are defined as the intersection of the body and head variables

frontier = vars(body) ∩ vars(head)

These variables are crucial, we already mentioned them before saying they are the "transfer of information" because the values that satisfy the body are transferred to the head
Unlike existential variables, frontier variables do not create new values.

They play an important role in more complex and restricted languages such as frontier-guarded and warded so it is an important term.

For Example:

A(x) & R(y) → B(y)
y is a frontier variable because it appears both in body and the head of the rule

x is not a frontier variable because it appears only in the body of the rule

We write front(σ) for the set of frontier variables of a rule σ, that is:
front(σ) = var (body(σ)) ∩ var (head(σ))

2.2.3 Affected vs Non-affected Positions

Affected positions

Definition:
The set of affected positions is inductively defined as follows:
-if there exists a rule in the ruleset and a variable x ∈ existential variables of this rule that appears at a position π in the head of the rule, then position π is affected.
-if there exists a rule in the ruleset and a frontier variable x ∈ frontier existential variables of this rule that appears in the body of the rule only at affected positions, and x also appears in the head of the rule at position π, then position π is affected. [2]

That means when a rule has an existential variable in its head, then the position in which the existential variable appears is affected. Also, the propagation continues when a variable is frontier and appears only in affected positions in the body, then every position in the head that it appears becomes affected.

For example:

$R(x) \rightarrow \exists y\, S(x,y)$
y is existential variable
It appears in position S[2]
That means S[2] is affected because it can generate null values

Non-affected positions

Definition:
non-affected= positions\affected
The set of non-affected positions consists of all positions that do not belong to the set of affected positions [2]

These positions contain values from the original database or body variables that cannot get the null value.

In the same example as before:

$R(x) \rightarrow \exists y\, S(x,y)$
x is a body variable
It appears in positions R[1] and S[1]
That means R[1] and S[1] are non-affected because they cannot generate null values

Propagation intuition

It is very important to know that positions are transferred from rule to rule as explained in the affected positions definition and this makes calculating positions complicated when studying a ruleset with more than 1 rule.

Example 1:
$R(x) \rightarrow \exists y\, S(x,y)$
$S(x,y) \rightarrow T(y)$
We know from the first rule that S[2] is affected because y is an existential variable
When we study the second rule, S[2] remains affected and contains the variable y
y does not appear in any non-affected positions, it appears only in affected positions
That means all the positions in the head that contain the variable y will also become affected, so we conclude that T[1] is affected.

Example 2:
$P(x) \rightarrow \exists y\, R(x,y)$
$R(x,y) \,\&\, Q(y) \rightarrow A(y)$
$A(x) \,\&\, R(z,y) \rightarrow B(y)$
In the first rule R[2] is affected because y is an existential variable.
In the second rule R[2] is affected and contains the variable y but Q[1] also contains the variable y and it is a non-affected position.

Because y appears in a non-affected position that means A[1] will not become affected even though it is a head variable that contains the variable y.
On the contrary, in the third rule variable y appears only in an affected position (R[2]) and does not appear in any non-affected positions so that means B[1] will become affected.

This distinction is crucial because affected positions are “dangerous” and they are the ones causing the infinite data generation problems, the purpose of creating constrained languages is to limit them or make them unable to produce infinite nulls with some extra restrictions. On the other hand, non-affected positions are considered “safe”.

2.2.4 Variable Classification

In the context of TGD analysis, there are categories for variables based in the positions they appear and the way they are affected by existential variables. This classification is very important for the definitions of Guarded Family languages and express how dangerous each variable is in creation of null values. We only categorize body variables, variables that appear only in the head of a rule and not in the body do not categorize into these categories. There are 3 categories in total:

-Harmless Variables

Definition:
A variable x is harmless if at least one occurrence of it appears in the body at a non-affected position. [2]

That means at least one occurrence of the variable is in a “safe” position. Harmless variables are considered safe and they cannot produce null values.

-Harmful Variable

Definition:
A variable x is harmful if it is not harmless. [2]

That means it does not appear in a non-affected position so all of it is occurrences in the body are in affected position.

They are “suspicious” as there is a chance that they depend exclusively on values generated by existential variables.

-Dangerous Variables

Definition:

A variable x is dangerous if it is harmful and frontier variable. [2]

That means that it appears only in non-affected positions in the body and is carried to the head. They are the most critical category as they depend on null values and at the same time carrying these null values to the head. They can cause chain propagation of nulls and lead to non-terminality or data explosion, so they are the biggest threat to our goal which is to extract information from databases.

The variables are categorized in each rule separately. That is, for each rule the categorization must be done and it is important to note, to avoid any confusions, the variables are independent in each rule. For example, x can be harmless in one rule and dangerous in another rule, only the affected and non-affected positions are propagated and remain throughout the dataset. Before we begin categorizing variables, we must first calculate the affected and non-affected positions of the whole ruleset.

Example that includes all 3 categories:

$P(x) \rightarrow \exists y\, R(x,y)$

$R(x,y) \,\&\, Q(y) \rightarrow A(y)$

$A(x) \,\&\, R(z,y) \rightarrow B(y)$

$R(x,y) \rightarrow A(x)$

First we calculate the affected and non-affected positions which are:

Aff: { R[2], B[1] }

Non-aff: { P[1], R[1], Q[1], A[1] }

After that we process each rule separately

R1:

Variables: {x}

x appears in a non-affected position → x is harmless

R2:

Variables: {x,y}

x appears in a non-affected position → x is harmless

y appears in a non-affected position → y is harmless

R3:

Variables: {x,z,y}

x appears in a non-affected position → x is harmless

z appears in a non-affected position → z is harmless

y appears only in affected positions (R[2], B[1]) and is frontier → y is dangerous

R4:

Variables: {x,y}

x appears in a non-affected position → x is harmless

y appears only in affected positions but is not frontier → y is harmful

2.2.5 Datasets and language properties

In addition to studying individual rules, sets of rules which are referred to as datasets or rulesets are examined. A dataset is a set of rules Σ and its classification in a language of the Guarded Family depends on whether all of its rules satisfy the properties of the language.

More specifically, a ruleset/dataset belongs to a language (e.g. Guarded) if and only if every rule in the datasets belong to the Guarded TGDs, if there is even one rule that does not belong to the language then it is enough to violate the definition for the entire dataset not to belong to the language.

2.3 Linear TGDs

The least expressive and simplest constrained language.

Definition:

If body of a rule σ contains a single atom, then rule σ is called linear. [4]

Rules that are Linear:

$R(x) \rightarrow S(x)$

$S(x,y) \rightarrow T(y)$

$S(x,y) \rightarrow T(y) \ \& \ S(x)$

Rules that are not Linear:

$A(x) \ \& \ B(x) \rightarrow C(x)$

$A(x) \ \& \ S(x,y) \rightarrow C(x,y)$

Intuition

Linear rules are the simplest in terms of structure as they do not accept combining information from multiple atoms and each rule is based on a single tuple

This means that different relations have no joins.

Relationship with nulls and propagation

In linear rules propagation of null values is more controlled than in general TGDs but with that comes less expressiveness. The reason is that each rule is triggered based on a single atom in the body and it limits the way in they way values are combined when applying the rules. Joins between the rules lead to more complex and uncontrolled propagation of null values but they do not exist here.

As a result null values are generated through existential variables, but their propagation follows a simple linear path.

For example:

$R(x) \rightarrow \exists \ y \ S(x,y)$

$S(x,y) \rightarrow T(y)$

The null value of y is transferred from S to T without requiring combination with other data. Consequently, although the creation of null values in not avoided, their propagation is predictable and less prone to a very big growth of the generated data.

Why are they important

-They are the most basic language in the Guarded Family.

-They are used as a reference for more complex languages.

-They ensure better computational behavior compared to general TGDs reducing complexity but the simplicity of the language makes it have the smallest percentage of datasets that belong to it.

-If a rule is linear, then automatically it belongs to every constrained language in the Guarded Family.

2.4 Guarded TGDs

One of the most important and widely studied classes of restricted TGDs.

Definition:

A TGD σ is guarded if in body(σ) there exists an atom a such that
var (a) = var (body(σ)) [4]

More formally, a rule of the form:
A1,A2,…,An → B1, B2,…,Bn
Is guarded if there is an atom Ai such that every variable that appears in the body, also appears in Ai. That atom is called guard atom

Rules that are Guarded:

R(x) → S(x)
R is the only atom and the guard atom

S(x,y) → ∃z T(z) & S(x)

S is the only atom and the guard atom. The variable z does not play a role since it does not belong to the body

$S(x,y) \& R(x) \& P(y) \rightarrow S(x)$
S in the guard atom that contains all body variables

Rules that are not Guarded:
$A(x) \& B(y) \rightarrow C(x)$
There is no atom containing all body variables (x,y)

$A(x) \& S(x,y) \& R(z) \rightarrow C(x,y)$
There is no atom containing all body variables (x,y,z)

Connection with Linear
Linear rules are a special case of Guarded rules. If there is only one atom in the body then that atom automatically contains all the body variables and it is the guard atom. That means every Linear rule is also Guarded, but the reverse is not true.

Relationship with nulls and propagation
Guarded TGDs allow more expressiveness than linear rules because the body can contain more than one atom while the guard atom controls the propagation of null values.

The propagation of new null values introduced by the existential variables is limited, because having all body variables connected in one atom means that at least some of the values used come from consistent and related source, it prevents random or unrelated data to combine when applying the rule.

Unlike linear rules, there are joins in the body, but the guard restricts it and limits the combinatorial explosion that can result from uncontrolled joins. Basically the form of locality that the guard imposes limits the spread of affected positions, so it helps control harmful and dangerous variables.

Theoretical significance

Guarded rules have been studied a lot in database and ontology because they offer greater expressiveness than Linear TGDs with good computational properties. They allow efficient query answering in many cases but not but not at a very satisfactory level, that is why there are even more complex languages (such as frontier-guarded and weakly-guarded) that use Guarded TGDs as their base.

2.5 Frontier-Guarded TGDs

Frontier-Guarded TGDs are a generalization of Guarded TGDs that allow more expressiveness while maintaining control over the propagation of null values in the same way.

Definition:
A TGD σ is frontier-guarded if there is a ∈ body(σ) with front(σ) ⊆ vars(a). [1]

Intuition
It is very similar to the Guarded rules but in contrast with them, where the guard must contain all the variables in the body but in Frontier-Guarded the guard only needs to cover the frontier variables. That is because we are not actually interested in all variables but only to those that carry information to the head and that is why Frontier-Guarded focuses only on them. The atom that covers all frontier variables is called guard atom

Rules that are Frontier-Guarded:
$S(x,y) \rightarrow \exists z\ T(z)\ \&\ S(x)$
S is the only atom and the guard atom, x is the only frontier variable so it is the only one we are interested in and it is contained in atom S

$R(x,z)\ \&\ S(y) \rightarrow T(x,z)$
The frontier variables are x and z, so we are interested only in them. Since there is an atom R that contains both x and z then that is the guard atom. y is a body variable but it is not frontier so it does not matter if it is in the guard atom or not

$R(x)\ \&\ S(x,y) \rightarrow \exists z\ T(x,z)$

Same as before x and y are frontier variables and S is the guard atom. It does not matter if any variable appears more than once in the body

Rules that are not Frontier-Guarded:

$R(x) \& S(y) \rightarrow T(x,y)$

The frontier variables are x and y but there is no atom that contains both

$A(x) \& S(x,y) \& R(z) \rightarrow C(x,y) \& P(z)$

There is no atom containing all frontier variables (x,y,z)

Connection with Guarded

Every Guarded rule is also Frontier-Guarded (because if an atom includes all body variables, it also includes all frontier variables). However, the converse is not true and that is why Frontier-Guarded are more expressive.

Relationship with nulls and propagation

The basic idea of Frontier-Guarded is that they control the propagation of null values through frontier variables.

Frontier variables are the ones that pass information from the body to the head so controlling them means that the propagation of values is controlled. While Guarded restricts all variables, Frontier-Guarded restricts only those that affect the result allowing more expressiveness with the same complexity.

Relation to joins

The Frontier-Guarded TGDs allow even more flexibility in joins because there can be a lot of atoms without coverage of all variables as long as there is a guard atom with all the frontier variables.

Importance of frontier variables

Here we can see the connection with variable classification. Frontier variables are directly linked to dangerous variables. Reminder a variable x is dangerous if it is harmful and frontier. That is why Frontier-Guarded restricts only frontier variables.

Theoretical significance

Frontier-Guarded TGDs extend Guarded TGDs while offering greater expressiveness but as we also conclude from the results of the thesis and from other studies there is not much difference between them. There continues to be a need for even more expressive languages and it is 1 of the 3 languages that form the basis of Weakly-Frontier-Guarded

2.6 Weakly-Guarded TGDs

Weakly-Guarded TGDs are a generalization of Guarded and Frontier-Guarded which allows even greater expressiveness while maintaining a controlled way of propagating values.

Definition:

σ is weakly-guarded if there exists an atom a ∈ body(σ) which contains (or guards) all the variables of var(body(σ)) that appear only at affected positions. [3]

Intuition

The basic idea of Weakly-Guarded rules is that we do not need to check all variables like Guarded and not all frontier variables like Frontier-Guarded) but only the variables that appear in dangerous positions which means variables that are harmful or dangerous. The harmless variables are safe so there is no need to protect them and by checking all frontier variables we also check some harmful frontier variables that are not dangerous, so unlike Guarded and Frontier-Guarded we don't guard harmless variables

It is the first language in the Guarded Family we see that has a direct relationship with variable classifications and to determine whether a rule belongs to this language, the ruleset must by analyzed first. That means calculating all affected and non-affected positions and categorizing variables in each rule. The atom that guards all the affected variables is called weak guard.

TGDs that are Weakly-Guarded:

$P(x) \rightarrow \exists y \ \exists z \ R(x,y,z) \ \& \ T(y)$

R(z,w,y) & P(x) & T(y) → T(z)

First we calculate affected an non-affected positions

Aff: { R[2], R[3], T[1]}

Non-aff: { R[1], P[1] }

Then for each rule we categorize the variables

R1: x=harmless

R2: w=harmful, x=harmless, y=harmful, z=harmless

Now we can check each rule if it belongs to the Weakly-Guarded TGDs

R1: Does not have any harmful variables → Weakly-Guarded

R2: w and y are harmful variables so they are the ones we are interested in. Both exists in atom Z so Z is the weak guard → Weakly-Guarded

R(x) → ∃y S(x,y)

S(x,y) & S(z,y) → T(y)

T(x) → Z(y)

Aff: { S[2], T[1]}

Non-aff: { R[1], Z[1], S[1]}

R1: x=harmless

R2: x=harmless, y=dangerous, z=harmless

R3: x=harmful

Now we can check each rule if it belongs to the Weakly-Guarded TGDs

R1: Does not have any harmful variables → Weakly-Guarded

R2: y is the only dangerous and S is the weak guard → Weakly-Guarded

R3: x is the only harmful anf T is the weak guard → Weakly-Guarded

TGDs that are not Weakly-Guarded:

P(x) → ∃y ∃z R(x,y,z) & T(y)

R(x,w,z) & P(x) & T(y) → T(z)

Aff: { R[2], R[3], T[1]}

Non-aff: { R[1], P[1] }

R1: x=harmless

R2: w=harmful, x=harmless, y=harmful, z=dangerous

Now we can check each rule if it belongs to the Weakly-Guarded TGDs

R1: Does not have any harmful variables → Weakly-Guarded

R2: w and y are harmful and z is dangerous so they are the ones we are interested in. There is no atom containing all of them → Not Weakly-Guarded

$R(x) \rightarrow \exists y\, S(x,y)$

$S(x,y) \,\&\, S(z,y) \rightarrow T(y)$

$T(x) \,\&\, S(z,y) \rightarrow Z(y)$

Aff: { S[2], T[1],Z[1]}

Non-aff: { R[1], S[1]}

R1: x=harmless

R2: x=harmless, y=dangerous, z=harmless

R3: x=harmful, y=dangerous, z=harmless

Now we can check each rule if it belongs to the Weakly-Guarded TGDs

R1: Does not have any harmful variables → Weakly-Guarded

R2: y is the only dangerous and S is the weak guard → Weakly-Guarded

R3: y is dangerous and x=harmful. There is no atom containing both → Not Weakly-Guarded

Relationship with nulls and propagation

Weakly-Guarded directly aim at controlling the propagation of null values

Harmful variables appear only on affected positions, so they are the vectors of null propagation.

The weak guard ensures that harmful variables are controlled within an atom and do not spread uncontrollably so the propagation of nulls remains controlled even in complex rules.

Relationship between Guarded, Frontier-Guarded and Weakly-Guarded

Every Guarded rule is both Frontier-Guarded and Weakly-Guarded because in Guarded rules, an atoms covers all body variables so it covers both frontier variables and harmful variables.

Guarded $\subseteq$ Frontier-Guarded
Guarded $\subseteq$ Weakly-Guarded

However, this type of union does not exist between Frontier-Guarded and Weakly-Guarded rules.

There are rules that are Frontier-Guarded but not Weakly-Guarded and rules that are Weakly-Guarded but not Frontier-Guarded.

Frontier-Guarded $\not\subseteq$ Weakly-Guarded
Weakly-Guarded $\not\subseteq$ Frontier-Guarded

This distinction shows that different languages of TGDs target different aspects of the problem and have different ways to control propagation.

Expressiveness and Computational Complexity

Weakly-Guarded TGDs have a significant increase in expressiveness compared to the previous categories as they allow greater flexibility in the structure of the rules and allow complex joins. This makes them have a good percentage close to real database applications and ontologies.

However, this increased expressiveness comes at a cost because the complexity of algorithms increases highly compared to more simple languages and query answering becomes slower. That is because computations of affected and non-affected positions and categorization of variables in each rule in very big and complex datasets is a difficult process that requires time.

2.7 Weakly-Frontier-Guarded TGDs

Weakly-Frontier-Guarded TGDs is a combination of Frontier-Guarded and Weakly-Guarded as it combines their definitions.

Definition:

A TGD σ is weakly-frontier-guarded if there exists an atom a ∈ body(σ) such that a contains all affected variables from front(σ). [1]

Intuition

The basic idea is that not all variables are dangerous, not all frontier variables are dangerous and the critical ones are those that are both harmful and frontier because the transfer information and null values to the head. The atom a that contains all the dangerous variables is called weak guard.

In addition, to determine whether a rule belongs to this language, the ruleset must by analyzed first, same as Weakly-Guarded.

TGDs that are Weakly-Frontier-Guarded:

$A(x) \rightarrow \exists z\ R(x,z)$

$R(x,y), E(u) \rightarrow T(y)$

Aff:{R[2], T[1]}

Non-aff: {R[1], A[1], E[1]}

R1: x=harmless

R2: u=harmless, x=harmless, y=dangerous

Now to determine if each rule is Weakly-Frontier-Guarded:

R1: Does not have dangerous variables

R2: y is the only dangerous variable and R is the weak atom, so the rule is Weakly-Frontier-Guarded

$Base(x) \rightarrow \exists p\ P(p)$

$P(x) \rightarrow A(x)$

$P(x) \rightarrow B(x)$

$P(x) \rightarrow C(x)$

$P(x) \rightarrow \exists d\ D(x,d)$

$A(x), B(y), C(z), D(x,y), E(u) \rightarrow T(x,y,u)$

Aff:{ B[1], A[1], P[1], D[1], T[1], D[2], C[1], T[2]}

Non-aff: {T[3], E[1], Base[1]}

R1: x=harmless

R2: x=dangerous

R3: x=dangerous

R4: x=dangerous

R5: x=dangerous

R6: u=harmless, x=dangerous, y=dangerous, z=harmful

Now to determine if each rule is Weakly-Frontier-Guarded:

R1: Does not have dangerous variables

R2: x is dangerous and P is the weak guard

R3: x is dangerous and P is the weak guard

R4: x is dangerous and P is the weak guard

R5: x is dangerous and P is the weak guard

R6: x and y are dangerous and they are together in atom D so D is the weak guard and the rule is Weakly-Frontier-Guarded.

The difference with Weakly-Guarded is that Weakly-Guarded would require variable z to be in the weak guard because it is harmful, but since it is not frontier Weakly-Frontier-Guarded does not require that. Additionally, the difference with Frontier-Guarded is that Frontier-Guarded would require variable u to be in the weak guard because it is frontier but since it is not harmful Weakly-Frontier-Guarded does not require that.

TGDs that are not Weakly-Guarded:

$A(x), B(y) \rightarrow \exists z\ R(x, y, z)$

$C(y) \rightarrow \exists w\ P(w)$

$R(x,w,z)\ \&\ P(w,y) \rightarrow P(y, z)$

Aff:{R[3], P[1], P[2]}

Non-aff: {B[1], R[1], A[1], R[2], C[1]}

R1: x=harmless, y=harmless

R2: y=harmless

R3: w=harmless, x=harmless, y=dangerous, z=dangerous

Now to determine if each rule is Weakly-Frontier-Guarded:

R1: Does not have dangerous variables

R2: Does not have dangerous variables

R3: y and z are dangerous variables. There is no atom containing both, y is in atom P and z in atom R, so the rule is not Weakly-Frontier-Guarded

$P(x) \rightarrow \exists y \exists z\ R(x,y) \& Z(z)$

$R(x,y) \& Q(x) \rightarrow A(y)$

$A(y) \& R(t,y) \rightarrow B(y)$

$B(y) \& U(w) \& Z(z) \rightarrow C(y,w)$

$C(y,w) \rightarrow D(y)$

$D(y) \rightarrow \exists u\ T(y,u)$

$T(u) \& C(y,w) \& F(w) \rightarrow G(u,y)$

Aff: {B[1], R[2], A[1], Z[1], G[1], G[2], D[1], T[1], T[2], C[1]}

Non-aff: {C[2], R[1], Q[1], P[1], F[1], U[1]}

R1: x=harmless

R2: x=harmless, y=dangerous

R3: t=harmless, y=dangerous

R4: w=harmless, y=dangerous, z=harmful

R5: w=harmless, y=dangerous

R6: y=dangerous

R7: u=dangerous, w=harmless, y=dangerous

Now to determine if each rule is Weakly-Frontier-Guarded:

R1: No dangerous variables

R2: y is the only dangerous variable and R the weak guard

R3: y is the only dangerous variable and R the weak guard

R4: y is the only dangerous variable and B the weak guard

R5: y is the only dangerous variable and C the weak guard

R6: y is the only dangerous variable and D the weak guard

R7: y and u are both dangerous variables but there is no atom containing both, so it is not Weakly-Frontier-Guarded

Relationship with nulls and propagation

Weakly-Frontier-Guarded want to control the propagation of null values using only the dangerous variables, they are the main carriers of null propagation and they must appear within a common atom. In this way creation and propagation of null values is allowed but the critical point which is "pass" into the result is controlled.

Relationship between Weakly-Frontier-Guarded and the rest of the Guarded Family

The Weakly-Frontier-Guarded family has a special place in the Guarded Family hierarchy as it is the most general and expressive form among the languages. In contrast to other languages, there is subset relation that allow us to conclude that a rule that is Weakly-Frontier-Guarded necessarily belongs to any other language in the family (as for example with Guarded rules that belong to Weakly-Guarded rules), because it has the least restrictive control. As a result, this category has the greatest expressiveness, allowing the formulation of more complex rules with multiple joins and this increased expressiveness is shown by the results of our experiments.

Linear ⊆ Weakly-Frontier-Guarded
Guarded ⊆ Weakly-Frontier-Guarded
Frontier-Guarded ⊆ Weakly-Frontier-Guarded
Weakly-Guarded ⊆ Weakly-Frontier-Guarded
Warded ⊆ Weakly-Frontier-Guarded

Computational Complexity

Increased expressiveness has a high cost, the number of possible rule applications increases but the propagation of nulls is harder to predict and it is the language with the higher computational cost among the Guarded Family languages.

2.8 Warded TGDs

Warded TGDs focus directly on controlling the dangerous variables with the idea of wardedness, which is to place all dangerous variables in one body atom, called the ward, and to restrict how this atom interacts with the rest of the body.

Definition:

A TGD σ is warded there are no dangerous variables in body(σ), or there exists an atom $\alpha \in body(\sigma)$, called a ward, such that:

(1) all the dangerous variables in body(σ) occur in α,and

(2) each variable of $var(\alpha) \cap var(body(\sigma) \setminus \{\alpha\})$ is harmless. [2]

In simpler terms, the definition says that for a rule to belong in the Warded TGDs there must be:

1)An atom that contains all the dangerous variables (called the ward)

2)All the remaining variables that appear in the ward (that is, those that are not dangerous) if they also appear in another atom in the body they must be harmless, if they do not appear anywhere else in the body they can be either harmless or harmful.

If there are no dangerous variables the rule belongs to the Warded TGDs.

Intuition

The basic idea of Warded TGDs is that dangerous variables depend on null values and transfer information to the head but it is enough to trap them all in one atom (ward), we must also check how they interact with the rest of the body atoms.

TGDs that are Warded:

$R(x) \,\&\, A(u) \rightarrow \exists y\, S(x,y)$

$R(x) \rightarrow \exists z\, S(x,y,z)$

$S(x,y,z) \,\&\, P(x,u) \rightarrow T(y,z)$

Aff:{S[2], S[3], T[1], T[2]}

Non-aff: {R[1], A[1], P[1], P[2], S[1]}

R1: x=harmless, u=harmless

R2: x=harmless

R3: u=harmless, x=harmless, y=dangerous, z=dangerous

Now to determine if each rule is Warded:

R1: No dangerous variables

R2: No dangerous variables

R3: y and z are dangerous variables so they must be contained in the same atom. S is the ward which contains the dangerous variables and there is another variable x in the ward that appears in another atom in the body which is harmless, so it is Warded

$P(x) \rightarrow \exists y \exists z\ R(x,y,z) \& T(y)$

$R(x,w,z) \& P(x) \& T(y) \rightarrow T(z)$

Aff:{R[2], R[3], T[1]}

Non-aff: {R[1], P[1]}

R1: x=harmless

R2: w=harmful, x=harmless, y=harmful, z=dangerous

Now to determine if each rule is Warded:

R1: No dangerous variables

R2: z is the only dangerous variable and R is the ward. There are other variables in the ward which are x and w, x also appears in another body atom so it must be harmless and it is, w is harmful but since it does not appear in another atom in the body it is okay. So the rule is Warded

TGDs that are not Warded:

$P(x) \rightarrow \exists y \exists z\ R(x,y,z) \& T(y)$

$R(x,w,z) \& P(x) \& T(w) \rightarrow T(z)$

Aff:{R[2], R[3], T[1]}

Non-aff: {R[1], P[1]}

R1: x=harmless

R2: w=harmful, x=harmless, z=dangerous

Now to determine if each rule is Warded:

R1: No dangerous variables

R2: z is the only dangerous variable and R is the ward. There are other variables in the ward which are x and w, x also appears in another body atom so it must be harmless and it is, same with w now it appear in another body atom so it must be harmless but it is harmful. So the rule is not Warded

$A(x), B(y) \rightarrow \exists z\, R(x, y, z)$
$R(x, y, z)\ \&\ A(z) \rightarrow \exists w\, A(w,z)$
$R(x, y, z) \rightarrow S(y)$

Aff:{R[1], A[1], R[3], A[2]}
Non-aff: {B[1], R[2], S[1]}
R1: x=dangerous, y=harmless
R2: x=harmful, y=harmless, z=dangerous
R3: x=harmful, y=harmless, z=harmful
Now to determine if each rule is Warded:
R1: x is dangerous and A is the ward. There is no other variables in the ward
R2: z is dangerous but it appears on 2 atoms R and A so the rule is not Warded
R3: No dangerous variables
So the dataset is not Warded because dangerous variables appearing in multiple atoms is not allowed but something important to note in this dataset is that it belongs to the Guarded TGDs but not in the Warded TGDs.

Relationship with nulls and propagation

Warded TGDs are specifically designed to control the propagation of null values. While in generic TGDs nulls can propagate without control via joins, in the Warded TGDs null propagation is strictly limited and dangerous variables are confined within the ward and their interaction with other body atoms is only with harmless variables. This means uncontrolled combination explosion is avoided and null propagation is predictable.

Relationship with the rest of the Guarded Family

Warded TGDs are closely related to other languages of the family although they introduce a different type of restriction.

In contrast to the other 3 expressive languages in the family (Weakly-Guarded, Frontier-Guarded and Weakly-Frontier-Guarded) the Warded is the only language of which the Guarded TGDs is not a subset, that means there are rules that are Guarded but now Warded like the example above. Also, the opposite does not apply since Warded is more expressive language than Guarded.

Guarded $\not\subseteq$ Warded
Warded $\not\subseteq$ Guarded

The only language that is a subset of Warded is Linear, that means only when a rule belongs to the Linear TGDs we can be sure it also belongs to the Warded TGDs.

Linear $\subseteq$ Warded
Warded $\not\subseteq$ Linear

The only language that Warded is a subset of is Weakly-Frontier-Guarded as it is the most expressive language in the family.

Warded $\subseteq$ Weakly-Frontier-Guarded
Weakly-Frontier-Guarded $\not\subseteq$ Warded

In relation to Weakly-Guarded and Frontier-Guarded, there is no direct relationship between them and Warded. Warded is more targeted compared to Frontier-Guarded but does not control frontier variables and does not check all harmful variables like Weakly-Guarded so there is no way to put any language as a subset of any other. The only thing they have in common is that they are all 3 subsets of Weakly-Frontier-Guarded

Weakly-Guarded $\not\subseteq$ Warded
Frontier-Guarded $\not\subseteq$ Warded
Warded $\not\subseteq$ Frontier-Guarded
Warded $\not\subseteq$ Weakly-Guarded

Computational Complexity

Warded TGDs achieve an extremely important trade-off here that makes , making them one of the most practically relevant languages within the guarded family.

In terms of expressiveness, Warded TGDs are strictly more expressive than Linear, Guarded and Frontier-Guarded while allowing complex joins and complex dependencies From a computational perspective, despite its high expressiveness the query evaluation remains efficient while controlled explosive data growth is avoided.

As a result, Warded TGDs offer significantly better computational behavior compared to Weakly-Guarded and Weakly-Frontier-Guarded TGDs, while maintaining higher expressiveness than the other Guarded Family languages.

In conclusion, Warded TGDs provide the best balances between expressiveness and efficiency within the Guarded Family.

Chapter 3

Implementation Details

3.1 System Overview / Architecture **36**

3.2 Preprocessing **38**

3.3 Rule Analysis and System Representation **38**

3.3.1 Parsing of the rules 38

3.3.2 Internal Representation of Rules 39

3.3.3 Representation of positions and their importance 40

3.3.4 Auxiliary variable analysis 41

3.3.5 Categorization of variables and GlobalWardedAnalysis 41

3.3.6 Role of the App.java and App2.java files 42

3.3.7 Overall significance of the stage 47

3.4 Classification Engine **48**

3.4.1 Overall operation of the mechanism 49

3.4.2 Linear Checker 49

3.4.3 Guarded Checker 50

3.4.4 Frontier-Guarded Checker 50

3.4.5 Weakly-Guarded Checker 51

3.4.6 Weakly-Frontier-Guarded Checker 51

3.4.7 Warded Check 52

3.5 Challenges and Implementation Difficulties **53**

3.5.1 Understanding and converting theoretical concepts 54

3.5.2 From local analysis to global ruleset analysis 54

3.5.3 Dataset Inconsistencies and preprocessing issues 55

3.5.4 implementation and consistency of data structures 55

3.5.5 Performance and efficiency 56

3.5.6 Validation and debugging 56

3.5.7 Overall assessment 57

3.6 Implementation Summary and Transition to Experiments **57**

3.1 System Overview / Architecture

The aim of our thesis is to analyze datasets that exist on the internet in order to reveal the percentage of datasets that belong to each language of the Guarded Family. To do this, a Datalog± ruleset analysis system was designed and implemented, aiming to automatically classify any ruleset into the Guarded Family languages. More specifically, the systems accepts as input a set of logical rules and it calculates the affected and non-affected positions, classifies the variables of each rule as harmless, harmful and dangerous and checks whether they satisfy the conditions to belong to the Linear, Guarded, Frontier-Guarded, Weakly-Guarded, Weakly-Frontier-Guarded and Warded languages.

To achieve this goal, the system was designed with a modular architecture, consisting of discrete processing stage. First, we must clarify that the program was written in Java and it is a total of 14 .java files, each stage is different .java file and performs a specific role while passing its result to the next file. This ensures a clear separation of responsibilities, easier maintenance and the possibility of future expansion.

In addition, to run the datasets in our program, there are 2 Python scripts that we will analyze further later.

The overall system operation flow follows the following stages:
1.Receiving the initial datasets
2. Preprocessing and normalization of the rules through clean_rules.py
3. Rule parsing
4. Rule analyzing
5.Language classification
6.Results export using batch_analyze.py

The first stage concerns the preprocessing of input data. The initial datasets came from researchers who conducted a similar study to mine contained syntax that is not directly compatible with our system is parser, such as full URLs as atom names, special symbols

and disjunctive rules. For this reason, a cleaning and normalization process was applied and the rules converted into a format suitable for analysis.

Next, each rule is then converted into an internal representation by the parser. Each rule is stored as a structure that includes the body, the head and the existential variables. Atoms work as separate objects containing the name of the predicate and its arguments, while the positions of the variables are captured through a special position structure, allowing to check for affected positions and null propagation analysis.

The core of the system is the classification mechanism, which consists of individual checkers, each of which implements the definition of a Guarded Family language. For example, to check if a rule belongs to the Linear TGDs the LinearChecker.java is based on whether the body of the rule contains exactly one atom or not while the GuardedChecker.java and the FrontierGuardedChecker.java are based on the existence of an atom that covers all relevant variables which the language requires to.

For more complex languages, such as Weakly-Guarded, Weakly-Frontier-Guarded and Warded, the ruleset needs to be analyzed first before we get to the language checkers. The system first calculates the affected and non-affected positions with the help of a fixpoint procedure, considering the existential variables and the propagation of frontier variables between rules. Based on this information, the variables of each rule are classified as harmless, harmful or dangerous and then the WeaklyGuardedChecker.java, WeaklyFrontierGuardedChecker.java and WardedChecker.java are applied to determine if the ruleset belongs to these languages or not.

Finally, the analysis can be seen at output files. The system produces a .txt file with analytical results for each dataset and an excel file containing an automated summary classification table for all datasets, this process is made using the batch_analyze.py which makes the system evaluation and comparison of results easy. The mass execution of the process is also automated via the same script which ensures consistency, speed and reproducibility of the experimental process.

Overall, the system architecture has an emphasis on reliability and scalability so it can still support analysis of large rulesets with 100% accuracy and can be easily extended for additional languages or techniques.

3.2 Preprocessing

Dataset preprocessing was not part of the original code implementation of the Java analysis system. During the development of the system, the datasets used for testing were in a format accepted by the system but the exact format of the experimental datasets was not yet known, so the implementation was designed to work on already normalized logical rules.

After the datasets were collected, it was clear that an additional preprocessing stage was necessary because the datasets collected had an unsupported form of rule. For this reason, a separate Python script was late created to lean and normalize the rules before they were given as input to this Java system.

Although in execution the pre-processing step is the first step, since the preprocessing step belongs to the experimental workflow rather than the architecture of the analyzer in Java, it is described in detail in Chapter 4, Section 4.2.

3.3 Rule Analysis and System Representation

After the pre-processing stage, the main implementation of the system in Java follows. It takes a dataset, analyses the rules and classifies the dataset and each rule separately in the Guarded Family of languages. The first stage involves rule parsing and converting them into an internal representation that is suitable for further processing, this makes the simple string to structured objects in java that can be analyzed algorithmically.

3.3.1 Parsing of the rules

The paring of the rules is implemented in the RuleParser.java class, it is responsible for converting each line from a string to a Rule object. That means the parser gets a rule in

string format as a rule and the output is a fully structured representation the includes the body, head and existential variables.
The parser supports the 2 forms of writing a rule, either body -> head or head :- body and understands from the symbols "->" and ":-" which one is the correct format. This ensures flexibility in input datasets and it appears in the experiments how practical it is because our sets are of the form ":-" but the program was originally written for rules of the form "->" , if this flexibility did not exists then there would have to be another preprocessing script to convert the rules from one form to another.

Next, an analysis of the atoms in each part happens. The splitting of atoms is done carefully so there are no errors in cases with multiple arguments or nested expressions and the logic behind that is that it takes into account the depth of the parentheses and separates atoms only at top-level separation points. An important element here is to locate the existential variables, the parser supports the declaration of existential variables with the symbol "∃" or the keyword "EXISTS" and it also supports multiple existential variables, allowing expression of the forms:
∃y R(x,y)
EXISTS y,z T(x,y,z)
These variables are stored separately to be used later in the analysis of affected positions and propagation of null values because the affected position start from them.

3.3.2 Internal Representation of Rules

After parsing, each rule is represented by the class Rule.java, it is the central data structure of the system and a Rule object includes:
-List of body atoms
-List of head atoms
-Set of existential variables

This class acts as the central storage point for all the information needed to parse a rule. The choice to use immutable structures (List.copyOf and Set.copyOf)) ensures that the rule is data cannot accidentality change during parsing, so it makes the system more reliable.

The individual atoms of a rule are represented through the Atom.java class. This class stores the name of the predicate and its arguments, the arguments are stored in a list with the same exact order as they appear in the rule. This is important because the position each variable appears must remain the same so the affected positions analysis can be correct, if the order changes, then everything changes and the analysis is wrong. The body and head are stores as List<Atom> while the existential variables are stored in a Set<String>, these allow efficient access and prevent duplicate entries.

This internal representation allows the system to process each rule in a structured way, the program works on objects that are clearly defined so the analysis and the maintenance are simplified and easy to understand.

3.3.3 Representation of positions and their importance

To support subsequent analysis of affected and non-affected positions, the Position.java class is used. This class represents a specific position of a predicate, in the format R[1] , S[2] and so on. Each Position objects consists of:

-The name of the predicate

-The pointer to the position of the argument

The reason for having a separate structure for positions is classifying variable and studying the propagation of nulls is based on exactly where each variable occurs and not whether it just occurs or not. Position objects are stored in HashSet collections that allow efficiency during the calculation of affected and non-affected positions. Therefore, the position-level representation is necessary and it makes the system accurate to the concepts of affected positions and dangerous variables.

Position.java implements the equality and hash methods in a safe manner and it is necessary on subsequent analyzes that are based on collections from positions without duplicates.

3.3.4 Auxiliary variable analysis

The RuleAnalyzer.java class works as a helper analysis tool on Rule objects. It does not have a role on language checkers but it provides basic information needed by other parts of the implementation. More specifically, the RuleAnalyzer calculates:

-The set of variables that appear in the body

-The set of variables that appear in the head

-The set of all rule variables

-The set of frontier variables

This class helps the code clean and reusable. Instead of repeating the same logic in many different parts of the program it is concentrated in a single auxiliary unit. A reliable and consistent source of data for each rule is variables serves as the foundation for the analysis is next stages.

3.3.5 Categorization of variables and GlobalWardedAnalysis

Another key structure belonging to the internal representation is VarKind.java, it is implemented as an enum. When parsing a rule, its purpose is to classify all the variables with the categories variables belong to are harmless, harmful and dangerous.

Using enums instead of simple strings makes the code better and reduces the chances of possible errors. In addition, it demonstrates a fundamental theoretical ideal of the task within the implementation and the categorization is used later for language checkers since the definitions of some languages are based on these.

In addition to the VarKind structure, the system includes the GlobalWardedAnalysis.java class which is responsible for a global analysis of the whole ruleset. More specifically it computes the set of all positions in the ruleset, the affected positions, the non-affected positions and the classification of variables for each rule.

To implement this process the checker uses structures of the form:
Map<String,List<Position>>, so each variable is linked to all positions in the body it appears in allowing the system to check if it appears at least once in a non-affected

position. In addition, a Map<String, VarKind> structure is used to store the final result for each variable if it is harmless, harmful or dangerous.

GlobalWardedAnalysis is a critical component for language checkers as it provides the necessary information for their implementation letting them use already prepared information

3.3.6 Role of the App.java and App2.java files

In addition to the files of the mechanism and the structure, the App.java and App2.java files also have an important role in the system, as they function as two execution functions of the program with each one having a different purpose.

App.java is the main version used in the scirpt for the processing of many datasets, it reads a ruleset file, parses al the lines, creates the corresponding internal structures and then produces a new file containg only the results for the whole dataset. The result of the dataset contain all the positions, the affected and non-affected positions and for each language there is a Yes/No if the dataset belongs to it or not. It focuses only on the result we want to see without showing the process behind the results and is suitable for minimizing storage space and easy further processing.

On the contrary, App2.java was used at the building process of the system for a more detailed study of individual datasets. Its difference from App.java is that it gives a detailed output for each rule separately and not only the overall dataset result. It includes for each rule the variables, their categorization, their positions and Yes/No if the specific rule belongs to each language. In this way, App2.java functioned as a debugging and checking tool to verify the correctness of the implementation and to find the exact spots a problem occurs. Due to the complexity of the analysis, the output is much larger than the dataset, so if this were used for our study, we would have huge files as results that would waste too much storage space on details that do not play any role in our study since we are only looking at the overall result.

We are going to show the difference between the 2 outputs for the same input file

The input dataset is:

B(X) -> EXISTS Z A(X,Z)

D(X) -> E(X)

E(X) -> F(X)

A(X,Y) & C(Y) & R(Z)-> P(X,Z)

G(X) & H(Y) -> I(X)

The App.java outpout is:

=== GLOBAL (Sigma) ===

Rules count: 5

All positions pos(Sigma): [B[1], A[1], A[2], R[1], P[1], P[2], I[1], H[1], G[1], F[1], E[1], D[1], C[1]]

Affected positions aff(Sigma): [A[2]]

Non-affected pos\aff(Sigma): [B[1], R[1], A[1], P[1], P[2], I[1], H[1], G[1], F[1], E[1], D[1], C[1]]

Is the ruleset Linear? No

Is the ruleset Guarded? No

Is the ruleset Frontier-Guarded? No

Is the ruleset Weakly-Guarded? Yes

Is the ruleset Weakly-Frontier-Guarded? Yes

Is the ruleset Warded? Yes

The App2.java outpout is:

=== GLOBAL (Sigma) ===

Rules count: 5

All positions pos(Sigma): [B[1], A[1], A[2], R[1], P[1], P[2], I[1], H[1], G[1], F[1], E[1], D[1], C[1]]

Affected positions aff(Sigma): [A[2]]

Non-affected pos\aff(Sigma): [B[1], R[1], A[1], P[1], P[2], I[1], H[1], G[1], F[1], E[1], D[1], C[1]]

Is the ruleset Linear? No

Is the ruleset Guarded? No

Is the ruleset Frontier-Guarded? No

Is the ruleset Weakly-Guarded? Yes

Is the ruleset Weakly-Frontier-Guarded? Yes

Is the ruleset Warded? Yes

=== Rule s1 ===

Original: B(X) -> EXISTS Z A(X,Z)

Parsed: Rule{body=[B(X)], head=[A(X,Z)], existentialVariables=[Z]}

All positions pos(s): [B[1], A[1], A[2]]

Affected positions aff(s): [A[2]]

Non-affected pos\aff(s): [B[1], A[1]]

Body vars: [X]

Head vars: [X, Z]

Frontier vars: [X]

Variable kinds: {X=HARMLESS}

Harmless vars: [X]

Harmful vars: []

Dangerous vars: []

Is Linear? Yes

Is Guarded? Yes

Is Frontier-Guarded? Yes

Is Weakly-Guarded? Yes

Is Weakly-Frontier-Guarded? Yes

Is Warded? Yes

=== Rule s2 ===

Original: D(X) -> E(X)

Parsed: Rule{body=[D(X)], head=[E(X)], existentialVariables=[]}

All positions pos(s): [E[1], D[1]]
Affected positions aff(s): []
Non-affected pos\aff(s): [E[1], D[1]]

Body vars: [X]
Head vars: [X]
Frontier vars: [X]
Variable kinds: {X=HARMLESS}
Harmless vars: [X]
Harmful vars: []
Dangerous vars: []

Is Linear? Yes
Is Guarded? Yes
Is Frontier-Guarded? Yes
Is Weakly-Guarded? Yes
Is Weakly-Frontier-Guarded? Yes
Is Warded? Yes

=== Rule s3 ===
Original: E(X) -> F(X)
Parsed: Rule{body=[E(X)], head=[F(X)], existentialVariables=[]}

All positions pos(s): [F[1], E[1]]
Affected positions aff(s): []
Non-affected pos\aff(s): [F[1], E[1]]

Body vars: [X]
Head vars: [X]
Frontier vars: [X]
Variable kinds: {X=HARMLESS}
Harmless vars: [X]
Harmful vars: []

Dangerous vars: []

Is Linear? Yes
Is Guarded? Yes
Is Frontier-Guarded? Yes
Is Weakly-Guarded? Yes
Is Weakly-Frontier-Guarded? Yes
Is Warded? Yes

=== Rule s4 ===
Original: A(X,Y) & C(Y) & R(Z)-> P(X,Z)
Parsed: Rule{body=[A(X,Y), C(Y), R(Z)], head=[P(X,Z)], existentialVariables=[]}

All positions pos(s):   [R[1], A[1], A[2], P[1], P[2], C[1]]
Affected positions aff(s): [A[2]]
Non-affected pos\aff(s): [R[1], A[1], P[1], P[2], C[1]]

Body vars: [X, Y, Z]
Head vars: [X, Z]
Frontier vars: [X, Z]
Variable kinds: {X=HARMLESS, Y=HARMLESS, Z=HARMLESS}
Harmless vars: [X, Y, Z]
Harmful vars: []
Dangerous vars: []

Is Linear? No
Is Guarded? No
Is Frontier-Guarded? No
Is Weakly-Guarded? Yes
Is Weakly-Frontier-Guarded? Yes
Is Warded? Yes

=== Rule s5 ===

Original: G(X) & H(Y) -> I(X)
Parsed: Rule{body=[G(X), H(Y)], head=[I(X)], existentialVariables=[]}

All positions pos(s): [I[1], H[1], G[1]]
Affected positions aff(s): []
Non-affected pos\aff(s): [I[1], H[1], G[1]]

Body vars: [X, Y]
Head vars: [X]
Frontier vars: [X]
Variable kinds: {X=HARMLESS, Y=HARMLESS}
Harmless vars: [X, Y]
Harmful vars: []
Dangerous vars: []

Is Linear? No
Is Guarded? No
Is Frontier-Guarded? Yes
Is Weakly-Guarded? Yes
Is Weakly-Frontier-Guarded? Yes
Is Warded? Yes

Therefore, both files serve the execution of the same system but they have different purposes. App.java is for massive and concise analysis of multiple datasets just to see the results while App2.java is for detailed investigation of a single dataset.

3.3.7 Overall significance of the stage

The parsing and internal representation stage is the basis on which the entire system was built. Without a reliable conversion of rules from string to data structures, parsing and internal representation stage the analysis of whether a dataset belongs to a language or not would not be possible.

The choice to have a lot of separate classes including classes for rule, atom, position, variable, categorization and more was made so the result would be a clean and organization code and most importantly to ensure a clear correspondence between the theoretical concepts and the practical implementation. In this way, the system has a clean structure, more scalability, easier maintenance and it makes it simpler to understand the outcomes and how they are produced.

3.4 Classification Engine

After parsing and the creation of the internal representation of the rules, the next stage, which is also what the system was created for, is the classification engine.

This stage is the core of the implementation, and it is responsible to check each rule separately if it belongs to a certain language and ultimately to determine whether a set of rules belongs to specific languages of the Guarded Family.

This implementation is based on a set of specializes checker classes, where there is one class for each language in the Guarded Family. More specifically, the following classes are used:
-LinearChecker.java
-GuardedChecker.java
-FrontierGuardedChecker.java
-WeaklyGuardedChecker.java
-WeaklyFrontierGuardedChecker.java
-WardedChecker.java

Each checker implements the logic of the theoretical definition we analyzed in the preliminaries and provides a method that accepts a rule (or a ruleset) and responds with a Yes or No whether the rule belongs to the language or not, this way the system has a clear mapping between the theoretical definitions and the implementation.

The classification process relies on the results of the GlobalWardedAnalysis, which provides affected positions and variable classification used by several checkers.

3.4.1 Overall operation of the mechanism

The classification mechanism is activated through App.java or App2.java files and for each dataset it:

-Reads all rules and converts them into objects.

-Calculates affected and non-affected positions.

-Calculates harmless, harm and dangerous variables.

-Applies languages checkers to each rule.

-Collects the results at the ruleset level.

-Initializes a Yes/No variable for each language which remains Yes only if all rules of the ruleset belong to that language and if at least 1 rule does not belong to the language, the variable becomes a No for the whole ruleset.

3.4.2 Linear Checker

The Linearity checker is the simplest of all and it is implemented in the LinearChecker.java class. The theory is that a rule is linear if its body contains exactly one atom and the implementation is based on that only, it checks the number of atoms in the body of the Rule object. In the code there is direct access to rule.getBody().

The logic is:

-If body.size()==1 -> return true

-Else -> return false

Reminder that this language is the least expressive of all languages and that is why no additional structure or analysis is required.

The existence of a separate class for the linearity maintains the consistency of the system, because it would be wrong to have separate classes for complex languages and not have them for simple ones, and allows easier understanding and extension of the system.

3.4.3 Guarded Checker

The guarded check is implemented in the GuardedChecker.java class and it is based on checking the coverage of variables through an atom (the guard atom).

In the code, this is implemented as follows:
-Uses RuleAnalyzer to extract all body variables
-Obtains the arguments for each atom in the body using atom.getArguments() and converts them into a set of variables
-Checks if the set of variables of the atom contains all body variables
-If it finds an atom that contains all the body variables then it returns true
-If it does not find such an atom, then it returns false

The implementation requires structures created in the parsing stage and Set is also used for fast checking to avoid duplicate occurrences of variables.

3.4.4 Frontier-Guarded Checker

The FrontierGuardedChecker.java class impliments the Frontier-Guarded check which is similar to the Guarded check but changes the set of variables examined.

In the code:
-The frontier variables are calculated through the RuleAnalyzer (Set<String> frontier = RuleAnalyzer.getFrontierVariables(rule)
-For each atom in the body the arguments are extracted with its variables
-Checks if an atom contains all the frontier variables
-If it finds an atom that contains all the frontier variables then it returns true
-If it does not find such an atom, then it returns false

The code is very similar in terms of structure to the GuardedChecker.java which shows the good reuse of the architecture since what they do is quite similar.

3.4.5 Weakly-Guarded Checker

The Weakly-Guarded check is implemented in the WeaklyGuardedChecker.java class. It is the first more complex language checker and it introduces the concept of affected positions, for this reason it accepts as an additional input the set affSigma. The affected positions for the whole ruleset are already calculated in the GlobalWardedAnalysis class.

The code:
-For each variable in the body, it defines its positions (using Position)
-Checks if any variable appears in an affected position since only variables appearing in affected positions are relevant for this check of Weakly-Guarded language (A rule is Weakly-Guarded if there is an atom in the body which contains all the harmful variables of the rule)
-Creates a new set affectedVars with all these variables
-For each atom in the body it checks if there is an atom covering all the variables in the affectedVars set (the weak guard)
-If it finds such an atom, then it returns true
-If it does not find such an atom, then it returns false

An important implementation detail in this checker is that it maps variables to positions via Positions objects and then intersection with affSigma.

3.4.6 Weakly-Frontier-Guarded Checker

Weakly-Frontier-Guarded checks for the weak guard that contains all variables that are frontier and harmful. The Weakly-Frontier-Guarded checker is implemented in the WeaklyFrontierGuarded.java class and it combines the concepts of the FrontierGuardedChecker.java and the WeaklyGuardedChecker.java.

The computation includes:
-Computation of the frontier variables
-For each frontier variable it checks if it appears in an affected position
-Creates a set relevantVars= frontier ∩ affected

-Checks if there is an atom that covers all the variables in the relevantVars set (the weak guard)
-If it finds such an atom, then it returns true.
-If it does not find such an atom, then it returns false

This language is the most expressive so that difficult part is that is requires double filtering from two different structures (variables and positions) that come from parsing and global analysis. Through this language, it is evident how important it is for all pieces of code to be separate and organized, but also how practical the code is when it reuses implementations that were made in the early stages at the implementation of language checkers.

3.4.7 Warded Check

The warded check is the most complex and important check in our system and it is implemented in the WardedChecker.java class. Unlike the other languages, here examining the existence of the guard is not enough, the variables within this atom must also be examined.

The implementation of course depends heavily on the results of the GlobalWardedAnalysis that has already calculated all affected and non-affected positions and categorized all variables, so the checker uses already prepared information.

The WardedChecker.java checks whether a rule satisfies the warded conditions using the GlobalWardedAnalysis object as input.

In the code:
-First of all, it receives a list with all the dangerous variables
-If the list is empty then the rule is Warded
-If there are dangerous variables, the checker examines each atom of the body if it contains all the dangerous variables (if it is a possible ward)

-For each such atom, a set with its variables is created and it is initially checked whether it contains all the dangerous variables, if it does not then the atom is rejected as a possible ward candidate.
-If the atom contains all the dangerous variables, then follows a second check. The system finds all the other variables, apart from the dangerous ones, appearing in the atom that also appear in the remaining atoms of the body. This calculation is done with sets of variables, it creates one set with all the atom variables and one set with all the remaining body atoms variables and takes their intersection. The critical point of the implementation is that every variable in that intersection must be harmless, so it receives from GlobalWardedAnalysis the set of harmless variables and checks whether each variable of the intersection set belongs to it.
-If even one variable that appears in the candidate ward and the rest of the body is not harmless, then the specific atom cannot be a valid ward atom, if every variable that appears in the candidate ward and the rest of the body is harmless then the atom is the ward and the rule becomes Warded.
-If no atom satisfies both conditions then the rule is not Warded.

This implementation clearly shows that Warded checking is more complex than other languages and the separation makes the code clean and closer to the theoretical formulation of the language.

3.5 Challenges and Implementation Difficulties

During the development of the system, we came across a lot of challenges both at the theoretical and the implementation level. Of course, these challenges had connections between them as any incorrect understanding of the theoretical concepts led to incorrect implementation and so many times some parts of the implementation were done 2 and 3 times. The most important challenges we encounter are presented below:

3.5.1 Understanding and converting theoretical concepts

The main challenge before starting the system implementation was to understand the theoretical definitions of the Guarded Family languages and slowly try to convert them into an algorithm.

This part of Computer Science is something we have not dealt with before and so concepts like affected positions, frontier variables, dangerous variables, restricted languages etc. were completely new to us and more complex than typical programming problems. Our supervisor, Dr.Andreas Pieris, helped us a lot in this step. In many cases the initial understanding we had for some of the definitions was incorrect or incomplete and that resulted in the implementation of checks that had wrong output without us knowing it at the given moment.

This led to a repeated cycle of revision, where theoretical concepts where tested and re-examined, corrected and then re0implemented. This process was time-consuming, but it was necessary so that the final system can correctly reflect the theoretical concepts and give the correct output for any dataset, every stage of the implementation went through a lot of tests.

This difficulty shows the general challenge of turning theory into practice particularly in our system where definitions are abstract and bases on properties of entire sets of rules.

3.5.2 From local analysis to global ruleset analysis

Another major challenge for us was the transition from the local analysis of a single rule to the global analysis of the entire ruleset with the chained propagation that connects the rules.

At the start, the implementation was based on the independent processing of each rule but for the calculation of affected and non-affected positions (and later the implementation of language checkers), information that propagates between all the rules of the ruleset is needed, it is not enough to analyze the affected and non-affected position of each rule separately.

This led to the need to implement the GlobalWardedAnalysis.java class that was one of the most difficult classes. The propagation of affected positions process had to be done correctly as it is the way that the rules communicate with each other and influence each other.

Debugging this process was challenging, as we first had to analyze the rules on paper and then check if the system is results agreed with mine, a process that was time-consuming and difficult since it had to be done for many datasets with different properties. It was very important to be 100% sure that this part was correct before moving on to the next ones as it would be difficult to detect errors and the design of the architecture was influenced by this challenge that led to the separation between local and global analysis.

3.5.3 Dataset Inconsistencies and preprocessing issues

As mentioned before, the datasets used in this work were not originally in an acceptable form for our program and this created some issues. The most common were the use of full URLs as predicate names, the use of symbols “.”,”?” and other in names and disjunction rules.

These problems made direct parsing impossible so as a result we had to create a dedicated preprocessing script to normalize the datasets. However, designing this script was itself a challenge as it needed to handle more than 18000 datasets without altering the meaning of the rules. Also, by creating a new file for each dataset a storage problem occurred because we had about 37000 files with rulesets.

3.5.4 implementation and consistency of data structures

For the implementation of the overall analysis, we used complex data-structures, such as variable-positions mapping and variable categorization. More specifically, structures as:
Map<String, List<Position>>
Set<Position>
Map<String, VarKind>

These structures required a lot of consistency between them, for example each variable had to be correctly mapped to all the positions it appeared and these positions had to be correctly categorized as affected or non-affected. Any inconsistency here could lead to incorrected categorization and then incorrect result so any inconsistency between them leads to a chain of mistakes basically, that is why we needed to spent time to understand these structures how exactly they work in Java.

3.5.5 Performance and efficiency

Although performance was not the main goal of the work, it was a practical challenge especially while knowing this system needs to be able to process a large number of datasets.

The need to calculate the propagation of affected positions and after that to categorize variables and apply multiple language checks to each rule increases the computational cost and especially in datasets with over 10 thousand rules it becomes very complicated. In addition to that, repeated operations on structures we described before like sets and maps affected the overall execution time even more.

To address these issues, we made some choices to reduce complexity:
-Computing global information only once per dataset
-Using efficient structures like HashSet and HashMap
-Avoid unnecessary recalculations and recalling things already calculated

The system obviously still needs hours to run all the datasets analyzed and may be able to be improved even further in the future, however these choices allowed the system to be practical for mass processing and can be easily reused without further improvement.

3.5.6 Validation and debugging

Finally, another major challenge was to verify the correctness of the system. As we already mentioned this process had to be done at a lot of stages before continuing to the next one to make sure everything is okay, so we had meetings with our supervisor every time a completed a new stage to discuss it with him and we did a lot of testing on our own

of course which required to find that correct response on our own and then compare it with the program result. Due to the complexity of the logic, especially in the language checkers which were the hardest, it was not always easy to identify the source of the problem in the code.

That is the way App2.java was used in the building process of the system, it is a debugging tool and its ability to display detailed information about each rule and each rule is variables allowed easier step-by-step validation of correctness.

3.5.7 Overall assessment

Overall, the implementation of the system had a lot of challenges related to theory, data management and algorithmic complexity. The need to properly understand a lot of definitions, design a complex system based on them using efficient data structures and deal with non-ideal datasets made the process particularly demanding.

Despite the difficulties, the final system manages to integrate all the individual elements into a single and functional architecture, capable of analyzing big number of real datasets and producing reliable results.

3.6 Implementation Summary and Transition to Experiments

Chapter 3 presented in detail the implementation of the system and the way it works for analyzing and classifying rules in the Guarded Family of languages. Starting from the preprocessing stage, the dataset cleaning process was described as well as some problems occurred in this phase.

Then, the internal representation of the rules thought the basic data structures was presented, as well as the way variables and positions work in our system. Emphasis was given to GlobalWardedAnalysis class which is a key class in the implementation to transfer information to the language checkers.

In addition, the classification mechanism was analyzed, and we explained where each language is implemented and how. They way in which the theoretical definitions are

converted into algorithmic rules and applied to each rule and the way the total results for the datasets are collected were presented.

Finally, some main challenges that occurred during the development of the system were mentioned, both in terms of understating the theory behind the system and in terms of implementation.

With the completion of this chapter, the system is now fully implemented and ready for the datasets evaluation. The next chapter presents the experimental procedure, where the system is applied to the datasets collects and the conclusions regarding the frequency of each language are presented and analyzed.

Chapter 4

Experimental Evaluation

4.1 Datasets **60**

4.2 Data Preprocessing **60**

4.3 Experimental Setup and Reproducibility **63**

4.4 Execution Pipeline (batch_analyze.py) **68**

4.5 Experimental Results **73**

4.5.1 Total number of datasets 73

4.5.2 Distribution of datasets by language 74

4.5.3 Interpretation of results 74

4.5.4 Expressiveness comparison 76

4.5.5 Balance of expressiveness and complexity 77

4.5.6 Observation for real datasets 77

4.5.7 Visualization of results 78

4.5.8 Analysis based on dataset size 83

4.6 Discussion and Interpretation of Results **87**

4.6.1 Confirmation of the theoretical hierarchy 87

4.6.2 Behavior of real datasets 88

4.6.3 The role of null propagation 88

4.6.4 Syntactic vs Semantic restrictions 89

4.6.5 Limitations of aggregate analysis 89

4.6.6 Practical implications 89

This chapter presents the experimental evaluation of the proposed system for rule analysis. The data preprocessing procedure is described, aiming to transform initial datasets into a suitable for the system format for analysis and then the experimental environment and execution procedure are presented. In addition, the results are analyzed and the main conclusions are discussed.

4.1 Datasets

The datasets used in this experimental evaluation came from the Manchester OWL Corpus (MOWLCorp), a large publicly available ontology repository that has been used in previous research works similar to ours, related to ontology and reasoning.

The corpus includes thousands of ontology files in OWL/XML format and metadata describing some features of the ontologies. The datasets were not created for this work, but they were taken from this repository with the aim of studying the practical use and appearance of Guarded Family languages in real rulesets.

In total, the repository contains around 18700 datasets however they were suitable for analysis because they contained unsupported syntax and disjunctive rules so it was necessary to create a data processing script described in the next chapter to normalize them for our parser.

This corpus was chosen for our work because it is based on realistic and heterogeneous datasets that come from different domains and different engineering practices, so they provide an objective basis for us to study the occurrence of Guarded Family languages.

The datasets show significant variation in terms of their size and complexity, there are datasets with small and simple rules and there are large datasets with complex rules and joins, so this variation allowed us to study the behavior of guarded-based languages in datasets of different scales and draw conclusions for each scale separately.

4.2 Data Preprocessing

The first stage of the experimental workflow has to do with the preprocessing of the datasets we collected. The datasets used in this work are collected from different websites and they came from researchers who conducted a similar study as we mentioned earlier, we did not do any process to collect them. As a result, their syntax was not directly compatible with the parser that we developed so we needed to do some preprocessing to them.

More specifically, the original files contained full URLs as predicate names, special variables such as “?”, “.” and other symbols and disjunctive rules. Because our parser was designed to accept a simpler form of rules these elements created difficulties in parsing and could not be identified.

To address these issues, we created a Python preprocessing script named clean_rules.py. The purpose of the script is to clean and normalize the initial rules so our parser can recognize them before the main analysis. The script was a critical part of the overall process as it ensured that all data is written in a correct format so that the program does not produce incorrect results due to its syntax.

The cleaning process initially involved removing unnecessary blank lines and comments. Then, each rule was analyzed separately to correctly identify the body and the head of each rule. After that, to separate atoms and their arguments with safety, the script used a splitting logic that considered nesting of parentheses and angle brackets. This splitting method ensured that there are no incorrect splits on complex predicates.

The normalization of predicate names was probably the most important step, gull URLs were converted into simple, short predicate names using only the last part of the URL and not the website it came from etc. This made the rules more readable and fully compatible with our parsing program.

At the same time, variables were normalized. Variables declared with the symbol “?” where converted to simple name variables (e.g. ?X to X), for existential variables that were declared with the symbol “!” the way they are declared changed and became the classic way with the exists symbol with normalized names. This helped the algorithm clearly identify the existential variables with clear names.

In addition, the script detected and excluded rules with disjunctive existential rules. Disjunctive rules are rules of the form A(X) | B(X) :- C(X) (| means OR) that are defined for some of the languages we study, such as Weakly-Guarded and Linear but Warded is defined only for existential rules without disjunction. The removal of these rules was deemed necessary both for compatibility reason and because they do not belong to the

Vadalog / Warded Datalog± that our study focuses on. At the same time, the script performed a safety check just to be sure that no unsupported symbols or element are in the final cleaned file.

After this preprocessing process, for each dataset the script created a new .txt file with the clean version with the suffix _clean.txt, suitable for input into the Java program. These cleaned files were then used as the input of the Java analyzer during the experiments. This process made the implementation more stable and made sure that all files being processed are reliable without disjunctive rules, additional symbols, complex names or anything else that can cause problems to the parser.

Below is a small example of a file with rules before and after the cleaning process, to show to comparison between the original form and the normalized form

005192ad-16bd-484c-b01b-60829634c969_test.owl.owl (Original file)

<http://www.soton.ac.uk/~bt/ontology/ocean/test.owl#Person_CLASS>(?X) :-
<http://www.soton.ac.uk/~bt/ontology/ocean/test.owl#hasaddress_PROPERTY>(?X, ?Y) .

<http://www.soton.ac.uk/~bt/ontology/ocean/test.owl#Address_CLASS>(?X) :-
<http://www.soton.ac.uk/~bt/ontology/ocean/test.owl#hasaddress_PROPERTY>(?Y, ?X) .

<http://www.w3.org/2002/07/owl#Thing>(?X1) :-
<http://www.soton.ac.uk/~bt/ontology/ocean/test.owl#Person_CLASS>(?X1) .
<http://www.w3.org/2002/07/owl#Thing>(?X1),

<http://www.w3.org/2002/07/owl#Thing>(?X2) :-
<http://www.soton.ac.uk/~bt/ontology/ocean/test.owl#hasaddress_PROPERTY>(?X1, ?X2) .

<http://www.w3.org/2002/07/owl#Thing>(?X1) :-
<http://www.soton.ac.uk/~bt/ontology/ocean/test.owl#Address_CLASS>(?X1) .

005192ad-16bd-484c-b01b-60829634c969_test.owl.owl.xml.rules_clean.txt (Cleaned File)

Person_CLASS(X) :- hasaddress_PROPERTY(X, Y)
Address_CLASS(X) :- hasaddress_PROPERTY(Y, X)
Thing(X1) :- Person_CLASS(X1)
Thing(X1) & Thing(X2) :- hasaddress_PROPERTY(X1, X2)
Thing(X1) :- Address_CLASS(X1)

4.3 Experimental Setup and Reproducibility

The experimental evaluation of this work was carried out on our personal computer which is a MacBook Pro with an Apple M1 processor. The operating system used as macOS while the implementation was executed in VS code in a Java environment version "18.0.2.1". The system RAM memory is 8 GB. Recording these elements is important, as the performance of the analysis is affected by the size of the datasets, the total number of rules between and the computing resources of the system.

The experimental environment was chosen because Java is a language that we are very familiar with and the goal was to have full reproducibility of the results. Each step of the process was organized in a way that another user can understand the system and run it easily with the same data and collect the same results. For this reason, the overall process was organized in a specific folder structure which clearly separates the input data, the code and the results.

The basic structure of the experimental environment is this:

```
Ptyxiakh/
│
├── Ioannis_Michail_Thesis.docx
│
├── sources/
│   ├── TheSpace-Efficient Core of Vadalog [2].pdf
│   ├── Guarded-Based Disjunctive Tuple-Generating Dependencies [3].pdf
│   └── ...
│
```

```
├── unnormalized-translated-files/
│   ├── clean_rules.py
│   ├── dataset1.owl.xml.rules
│   ├── dataset2.owl.xml.rules
│   └── ...
│
├── datasets/
│   ├── dataset1_clean.txt
│   ├── dataset2_clean.txt
│   └── ...
│
├── vadalog-analyzer/
│   ├── batch_analyze.py
│   │
│   └── src/
│       ├── App.java
│       ├── RuleParser.java
│       ├── Rule.java
│       ├── Atom.java
│       ├── Position.java
│       ├── GlobalWardedAnalysis.java
│       ├── RuleAnalyzer.java
│       ├── VarKind.java
│       ├── LinearChecker.java
│       ├── GuardedChecker.java
│       ├── FrontierGuardedChecker.java
│       ├── WeaklyGuardedChecker.java
│       ├── WeaklyFrontierGuardedChecker.java
│       └── WardedChecker.java
│
└── results/
    ├── dataset1_clean_results.txt
    ├── dataset2_clean_results.txt
```

└── classification_summary.xlsx
└── classification_summary_help.xlsx

The unrnormalized-translated-files folder contains the original datasets before preprocessing and the clean_rules.py script which is used to normalize them. The cleaned datasets are stores in the datasets folder and the vadalog-analyzer folder contains the system code and the batch_analyze.py script. The results folder contains the file with the output results for each script and the total summary of the results classification_summary.csv file which became classification_summary.xlsx because below the automatically generated data we added the total numbers and calculated the percentage of each language through Excel functions. In addition, classification_summary_help.xlsx was created to create the graphs shown in Chapter 4.5.7.

Description of the execution process

The system execution for a big number of files is based on the main class App.java, which acts as the main class of the program. This class accepts as input a dataset file and an output path, it reads all the rules, converts them into an internal representation, performs the analysis and produces a result file stored in the output path. The system execution follows these stages:

1.Reading the input file

The program reads the file containing all the rules and ignores empty lines .

2.Parsing the rules

Each rule is converted from string to an object of the Rule class. This representation includes the body of the rule, the head of the rule and a set of existential variables.

3.Global analysis of rules

The system calculates the affected and non-affected positions for the entire ruleset using an iterative process. At first, the positions where existential variables appear in the head are considered affected, then through iterative steps, the propagation of these positions

through the frontier variables is examined over and over again, until no new affected position occur.

4.Variable categorization

Based on the affected and non-affected positions, the variables are categorized into harmless, harmful and dangerous.

5.Language checkers

For each rule, the corresponding checks for the languages are applied:

-Linear

-Guarded

-Frontier-Guarded

-Weakly-Guarded

-Weakly-Frontier-Guarded

-Warded

Each checker checks whether all the rules in the ruleset satisfy the corresponding theoretical definitions of each language.

7.Result generator

The system creates an output file containing the following:

-The number of rules

-The set of total positions

-The set of affected positions

-The set of non-affected positions

-The final classification for each language (Yes/No)

Execution for a single dataset

To execute the system on a single dataset the following commands are used:

javac *.java

java App input_clean.txt output_results.txt

The first argument is the input file

The second argument is the output file

Execution for a single dataset in analytical form

In addition to the main execution with App.java, there is also the possibility to execute the system with App2.java which is used for analysis purposes, debugging and verification of the correctness .

To execute App2.java on a single dataset the following commands are used:

javac *.java

java App2 input_clean.txt

In this case there is only one argument which is the input file and the output is displayed directly on the console .

The extra information displayed is the variables of each rule, their categorization, the positions each variable appears and the classification of each rule in all the Guarded Family languages.

Automated execution for multiple datasets (batch processing)

Due to the large number of datasets, the execution of the processing could not be done manually. For this reason, the batch_analyze.py script was created, as it automates the execution of the system for a large number of datasets.

There is a more detailed explanation of the script and how it works in the next chapter, Chapter 4.4.

To execute the batch_analyze.py script:

python3 batch_analyze.py

Data flow (Input → Output)

The overall flow of the experimental process is as follows:

Initial datasets

↓

clean_rules.py

↓

Cleaned datasets

↓

batch_analyze.py

↓

Java Analyzer (App.java)

↓

Result files

↓

Classification_summary.csv

Reproducibility of results

Reproducibility is ensured by the following features:

-All the dataset go through the same preprocessing file

-The same system is applied to all the datasets

-The execution is fully automated

-The results are automatically stores with a fixed structure

-The system is deterministic, that means it always produces the same output for the same input, it does not use random parameters or something that might affected the result.

Overall evaluation of the experimental environment

The experimental environment was designed to be reliable, stable, to have reproducibility and more importantly to be correct. The combination of preprocessing, a very good java analysis system and automated execution allow the processing of large number of datasets in a consistent manner and enables the extraction of reliable statistical results.

4.4 Execution Pipeline (batch_analyze.py)

After preprocessing the datasets and converting them into a clean format, the next stage of the process was the mass execution of the analysis system. Because the number of datasets was very large, the manual execution for each file individually was not practical. Manual processing could have created problems such as some data sets being left behind by mistake, some being executed 2 or more times, and the main problem would be that it would take a very long time to draw conclusions without easy reproducibility. For these

reasons, a second python script names batch_analyze.py was created, which functions as an automated experimental execution mechanism.

Batch_analyze.py does not change the content of the rules and does not perform any analysis on itself. The analysis remains the responsibility of the Java system, the script is role is to organize and execute the process in bulk:
-It locates all the cleaned datasets
-Calls the Java program for each of them
-Saves the results in a separate file
-Creates a centralizes csv file with the results of every dataset ran
In this way, the script functions as an automation layer of the process.

In the begging of the script, the basic folders and files that are used in the execution are defined:

```
DATASETS_DIR = Path("/Users/giannos/Documents/UCY/Ptyxiakh/datasets")
RESULTS_DIR = Path("/Users/giannos/Documents/UCY/Ptyxiakh/results")
SRC_DIR = Path("/Users/giannos/Documents/UCY/Ptyxiakh/vadalog-analyzer/src")

SUMMARY_FILE = RESULTS_DIR / "classification_summary.csv"
JAVA_CLASS = "App"
```

These variables determine where the cleaned datasets are located, where the results will be stored, the location of the Java code and which class will be executed between App.java and App2.java.

The first step of the execution is to locate all the cleaned files with the get_input_files function that selects only files that end with “_clean.txt”. In this way, the script ignores any other files in the folder and process only the files that are normalized from the other script, ensuring that the Java program will not be executed for incompatible datasets.

For each input file, the script automatically generates the corresponding output file name with the get_output_path function.

For example, if the dataset is named
dataset1_clean.txt
Then the corresponding output file will be named:
dataset1_clean_results.txt

This consistent naming makes it easy to link each output file to its corresponding dataset, both to the cleaned version and to the unnormalized version.

```
cmd = [
    "java",
    JAVA_CLASS,
    str(input_path),
    str(output_path)
]
```

This command essentially corresponds to executing
java App input_path output_path.

That is, for each dataset that the script calls App.java, it gives the cleaned version as the first argument and the output file with the results as the second argument. The execution is done with subprocess.run, with the working directory being the folder containing both the Java code and the batch_analyze.py script.

```
completed = subprocess.run(
    cmd,
    cwd=SRC_DIR,
    capture_output=True,
    text=True
)
```

The use of cwd=SRC_DIR here is important because it ensures that the Java program is executed from the correct directory, capture_output=True means that the script can save the standard output and standard error in case of a problem in execution.

Another important thing inbatch_analyze.py is that it does not simply stop on an error, if the Java program returns a non-zero exit code or does not create an output file, the script creates a special error result file and then continues the process for the rest of the datasets. This is done with the write_error_result function, which writes the error status, the dataset name and the error message to the output file. In this way problematic datasets are not lost from process but are recorded and can be checked later without interruption the whole process.

In a batch execution of thousands of datasets, it is possible to encounter some files with problems, if the file stopped the whole process every time it encountered one then it would be an issue because every now and then the process would have to be restarted from the beginning until all the files were fixed. On the other side, if it just skipped the files without saving the error files, it would not be possible to know which files were problematic and why, so fixing them will be very difficult and also the study would not be correct if you did not know why some rulesets do not run.

After executing each dataset, the script reads the generated result file and extracts the information needed for the csv file with the parse_result_file function. These functions created a dictionary with the basic fields: file, status, rules, linear, guarded, frontier_guarded, weakly_guarded, weakly_frontier_guarded, warded and error_message Each field is a columns in the final csv file, the file holds the name of the dataset, status indicates whether the execution was successful or not and returns the words “ERROR” or “OK”, rules holds the total number of rules in the dataset, error_message shows an error message if there is and the other fields have the classification for each language and show Yes/No.

For this to happen, the function reads the result file line by line and identifies specific phrases produced. For example:

```
elif line.startswith("Rules count:"):
    result["rules"] = line.split(":", 1)[1].strip()

elif "Is the ruleset Linear?" in line:
    result["linear"] = extract_yes_no(line)
```

```
elif "Is the ruleset Guarded?" in line:
    result["guarded"] = extract_yes_no(line)
```

This is the way that the script converts the output of the Java system into structured data to pass it to the csv file later without any chance of any mistake being made
The final step is to create the classification_summary.csv file in the end and that is done with the write_summary_csv function which writes all the results there.

This file is the basis for the experimental analysis of the results. Each row corresponds to a dataset and each column to information needed for the evaluation, so this file does all the work so that we have a ready-made analysis of results that is created automatically and quickly. Also, things that are missing from the automated file can be calculated very easily, such as the percentages of each language in relation to the total number of datasets that were successfully analyzed.

Overall, the execution pipeline of the batch_analyze.py can be described like this:
Cleaned datasets (_clean.txt)
↓
batch_analyze.py
↓
Call App.java for each dataset
↓
Generate separate result file for each dataset with result.txt at the end
↓
Read and export Yes/No classification results for every language
↓
Create classification_summary.csv

This process helped the experimental evaluation because it allowed the system to run for a large number of datasets quickly and without human error in a consistent manner. Without this script the execution of each dataset would have to be done manually and also

the collection of data and creation of the csv file would also have to be done manually, a process that would have been time-consuming and error-prone.

Furthermore, batch_analyze.py contributes to the reproducibility of the work, someone with the same folder and the same files can repeat the process and get the same results easily, or it can be used for other datasets easily because the program was not created only for specific data but must be able to manage any data with ease.

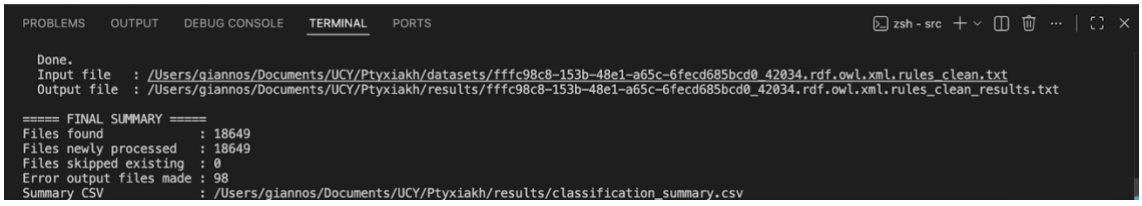
Finally, the batch_analyze.py script is not just a helper script, it is a core part of the experimental infrastructure that connects the datasets to the Java system, organizes the batch execution and produces the summary file which the analysis of the results is presented in the following section

4.5 Experimental Results

This section presents the results of the experimental evaluation of the system that emerged from the execution process described. In total, 18551 datasets were analyzed and used as input to the analysis system after preprocessing.

The results were organized in summary file classification_summary.csv which contains for each dataset the number of rules, execution status and classification of the dataset in each of the 6 languages in the Guarded Family. The extension changed to .xlsx after manually adding rows calculating the total number of datasets that belong to each language and their percentage.

4.5.1 Total number of datasets



Out of a total of 18649 datasets:

18551 datasets were successfully analyzed (status=OK)

98 datasets encountered an error during execution

The analysis below is based only on datasets that were successfully analyzed so that the results are reliable and no extra data from files with errors is taken into measure.

4.5.2 Distribution of datasets by language

The table below presents the distribution of datasets by all languages of the Guarded Family

Language	Number of Datasets	Percentage
Linear	11271	60%
Guarded	12952	69%
Frontier-Guarded	13247	71%
Weakly-Guarded	16659	89%
Weakly-Frontier-Guarded	16988	91%
Warded	15880	85%

The percentages were calculated as follows:

Percentage = (Total datasets of the language / Total datasets) * 100

4.5.3 Interpretation of results

The results presented in the previous table reveal important properties of real datasets in the web in relation to the Guarded Family languages. There is an expected trend for the percentage of datasets belonging to each language to increase as we move up the Guarded Family language hierarchy, which if it did not exist then the results could not have been correct in accordance with the theory.

Specifically, 60% of the datasets belong to the Linear language while Guarded and Frontier-Guarded datasets increase to 69% and 71% respectively. The significant increase is shown at the other 3 languages, Warded, Weakly-Guarded and Weakly-Frontier-Guarded which reach 85%, 89% and 91% of the datasets. This shows that most real datasets do not satisfy the strict constrains of less expressive languages but a high percentage of them can be included in more expressive languages based on the semantics of affected positions.

Some observations about each language:

Linear:

The linear language appears in 60% of the datasets and although it is not a low percentage it indicated that is the most restricted class. This is expected as the definition of Linear requires that the body of each rule contains only one atom which is not common in complex datasets but it is still a useful language for some small and simple datasets.

Guarded and Frontier-Guarded:

Guarded and Frontier-Guarded show higher percentages with Guarded having 69% and Frontier-Guarded 71%. The difference between them is due to the fact that Frontier-Guarded is a bit more relaxed language as it requires coverage for only frontier variables not all variables but both of them do not achieve a satisfactory percentage.

Weakly-Guarded:

Weakly-Guarded is the second most expressive language and something that is apparent here and in Weakly-Frontier-Guarded and Warded languages is that most datasets satisfy looser constraints based on affected positions and not just on syntactic features .

Weakly-Frontier-Guarded:

Weakly-Frontier-Guarded has the highest percentage with 91%, making it the most expressive language among those studied. This is expected as it is at the top of the Guarded Family hierocracy as it combines the loosest restrictions on both variables and affected positions.

However, the fact that a language covers the largest percentage of datasets does not necessarily mean that it is the most suitable for practical use. The great complexity of the language that comes with the increased expressiveness in query processing and evaluation make makes the language difficult to use

Warded:

A very important result of our work concerns the Warded language and it is the main finding of this thesis.

85% of the datasets belong to the Warded language, it is a very high percentage close to the percentages of Weakly-Guarded and Weakly-Frontier-Guarded, only a small percentage (~5%) of datasets are "lost" due to the additional constrains in comparison to Weakly-Guarded and Weakly-Frontier-Guarded, however Warded is significantly more restrictive than these two languages. This result is particularly important and can be interpreted as follows:
Warded seems to achieve a balance between expressiveness and restriction. On the one hand, it is expressive enough to cover the vast majority of real datasets and on the other hand, it introduces specific restrictions on the propagation of dangerous variables, which reduce a bit the complexity of the evaluation.

Unlike Weakly-Guarded and Weakly-Frontier-Guarded which rely only on the existence of a weak guard, Warded imposed additional restrictions to the behavior of variables within the ward, those restrictions limit the uncontrolled propagation of null values.

The fact that Warded covers 85% of the datasets shows that these restrictions it imposes on the variables of the ward are not too script for real data. On the contrary, it seems that most datasets already follow a structure that is compatible with the rules of the Warded language, so these ward restrictions actually make the language control the null propagation without losing much expressiveness.

4.5.4 Expressiveness comparison

The results confirm the theoretical hierarchy of languages
Linear ⊆ Guarded ⊆ Frontier-Guarded ⊆ Weakly-Frontier-Guarded
Linear ⊆ Guarded ⊆ Weakly-Guarded ⊆ Weakly-Frontier-Guarded
Linear ⊆ Warded ⊆ Weakly-Frontier-Guarded
As we move towards mor expressive languages, the percentage of datasets that belong to them increases in accordance with the theory.

4.5.5 Balance of expressiveness and complexity

One of the most important findings of the experimental evaluation is the behavior of the Warded language. Although Warded has strong restrictions , even stronger than Weakly-Guarded and Weakly-Frontier-Guarded TGDs, it still covers approximately 85% of the analyzed datasets. This percentage is remarkably high considering that the language imposes additional control on dangerous variables and null propagation.

The results suggest that Warded achieves an effective balance between expressive power and computational control. On the one hand, restrictive languages such as Linear and Guarded do not capture a large portion of real-world datasets because they impose strong limitations on rule structure. On the other hand, highly expressive languages such as Weakly-Guarded and Weakly-Frontier-Guarded allow more datasets but introduce increased reasoning complexity.

Warded appears to provide a practical middle ground between these two sides. Its restrictions are strong enough to limit uncontrolled propagation of null values, while still being flexible enough to model the majority of real-world rulesets.

This observation is particularly important for practical applications of ontology reasoning and rule-based systems, where both expressiveness and efficient query answering are necessary.

4.5.6 Observation for real datasets

Another important observation from the experiments is that real-world datasets are not completely arbitrary in their structure. The high percentages observed across all Guarded Family languages show that most rulesets are practical and already satisfy some form of structural or semantic restriction.

Even highly restrictive languages such as Linear and Guarded achieve relatively high percentages compared to what might be expected for unrestricted logical rulesets. This implies that rules used in real-world applications are often developed in an organized manner to prevent unnecessary joins and uncontrolled information spread.

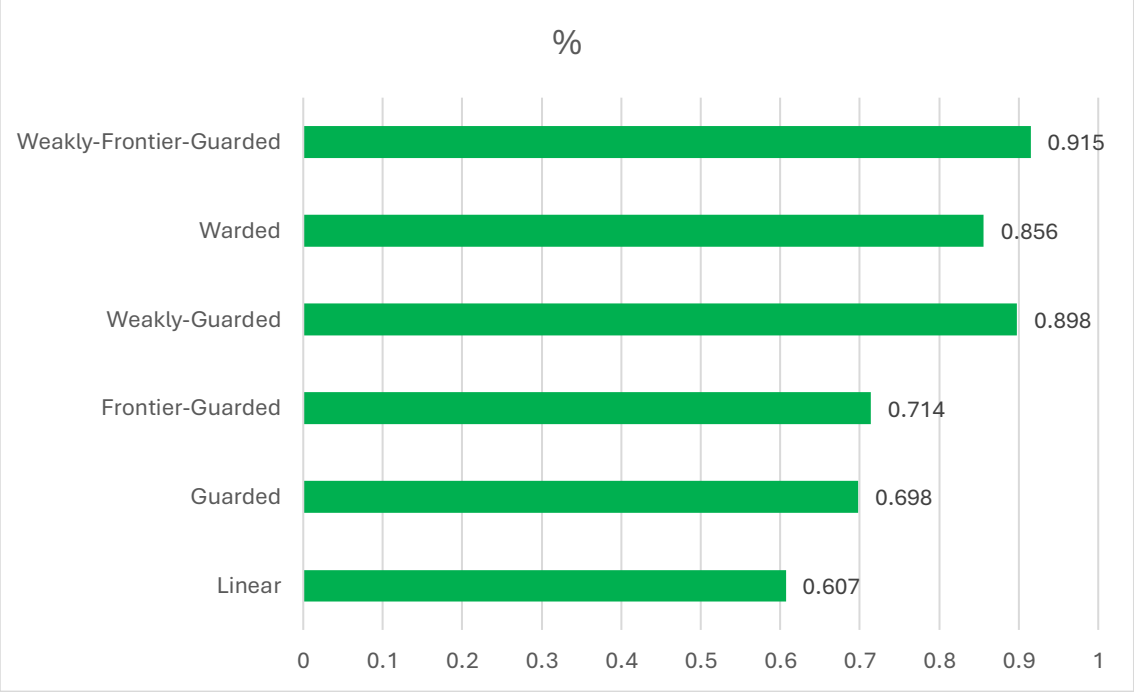
At the same time, the significantly higher percentages achieved by Warded, Weakly-Guarded and Weakly-Frontier-Guarded indicate that real datasets frequently require more expressive forms of reasoning and more flexible rule interactions.

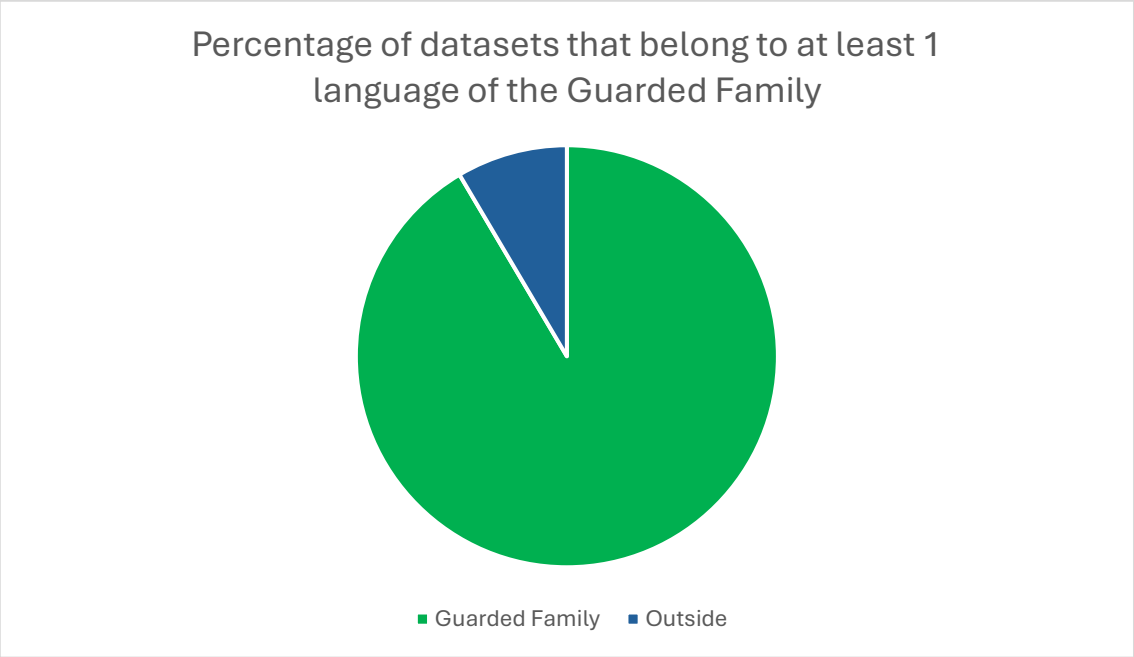
Overall, the experimental findings suggest that real world datasets tend to have a natural balance between expressiveness and structural control. This might help to explain why languages that manage harmful variables and null propagation are especially efficient at modeling real-world rulesets.

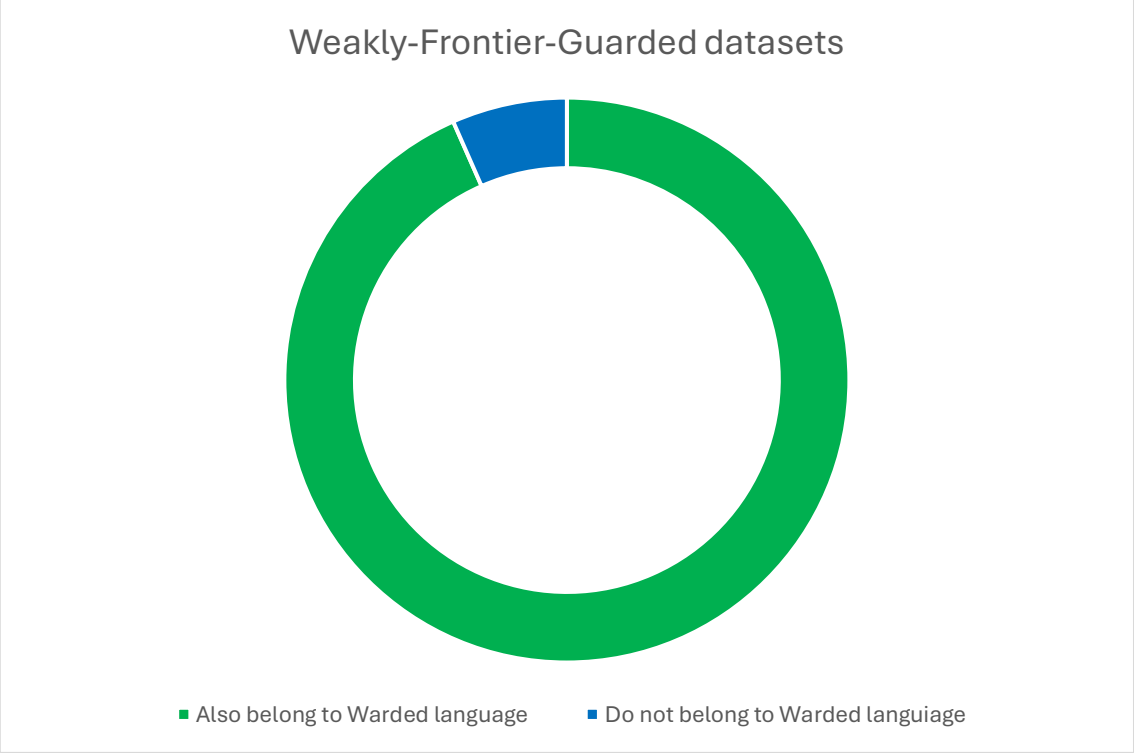
4.5.7 Visualization of results

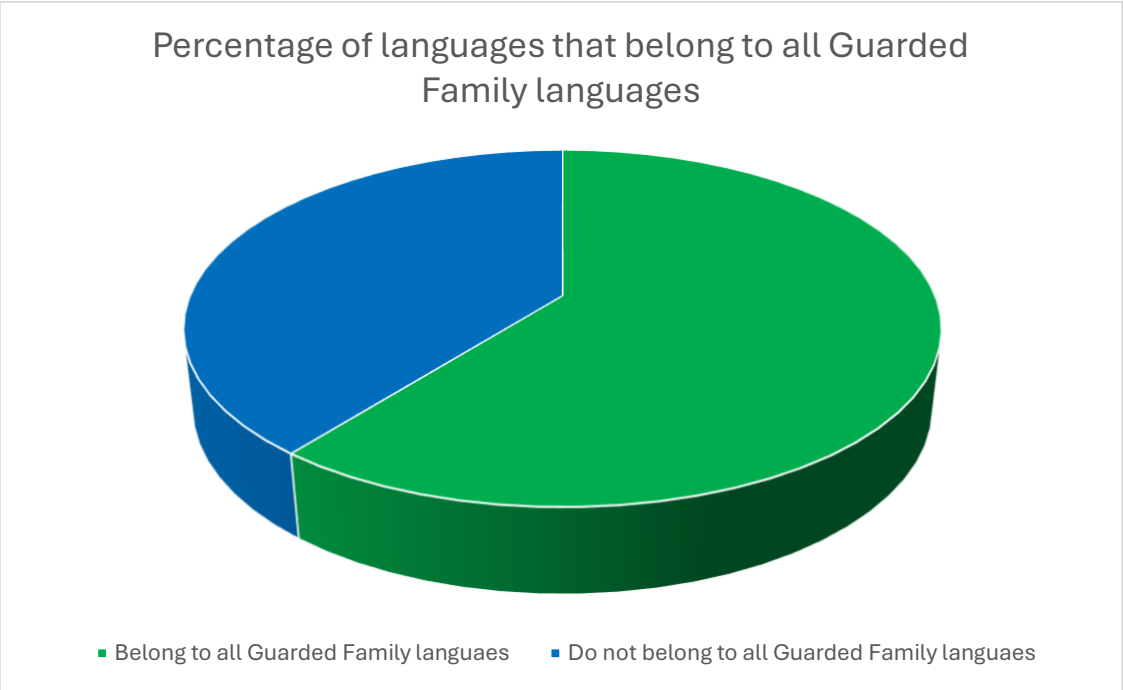
For better understanding of the results, we will show some graphs and pie charts that present them.

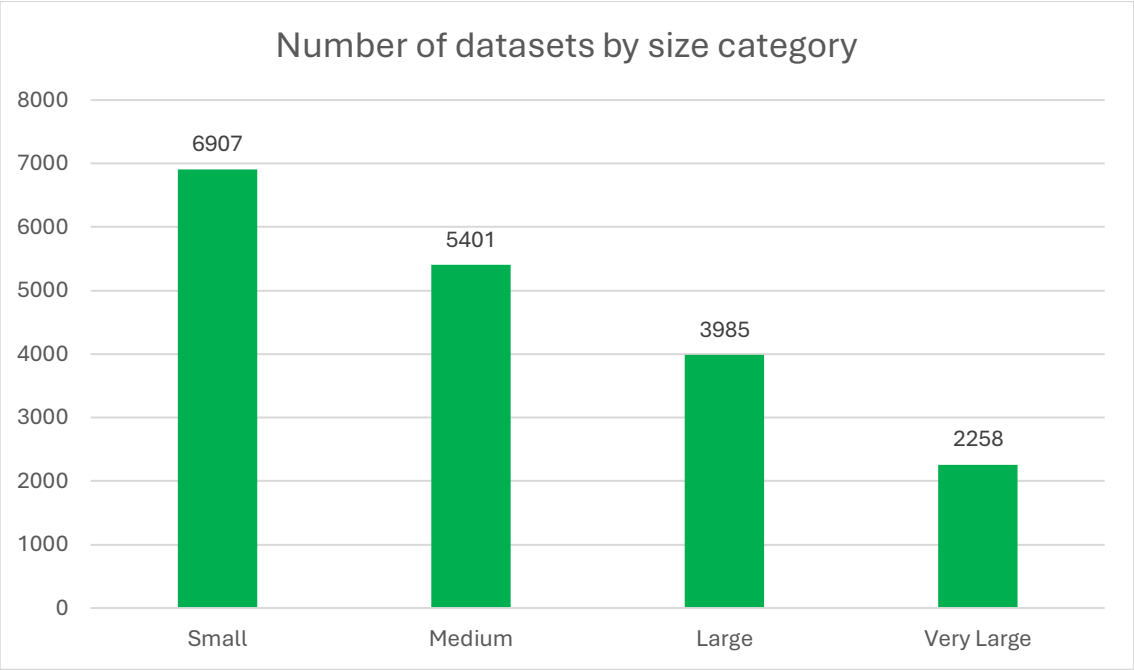
Also, there are graphs and an extra analysis follows bases on the size of the datasets as they will be categorized in 4 categories:
Small datasets (1–10 rules)
Medium datasets (11–200 rules)
Large datasets (200–1000 rules)
Very Large datasets (1000+ rules)

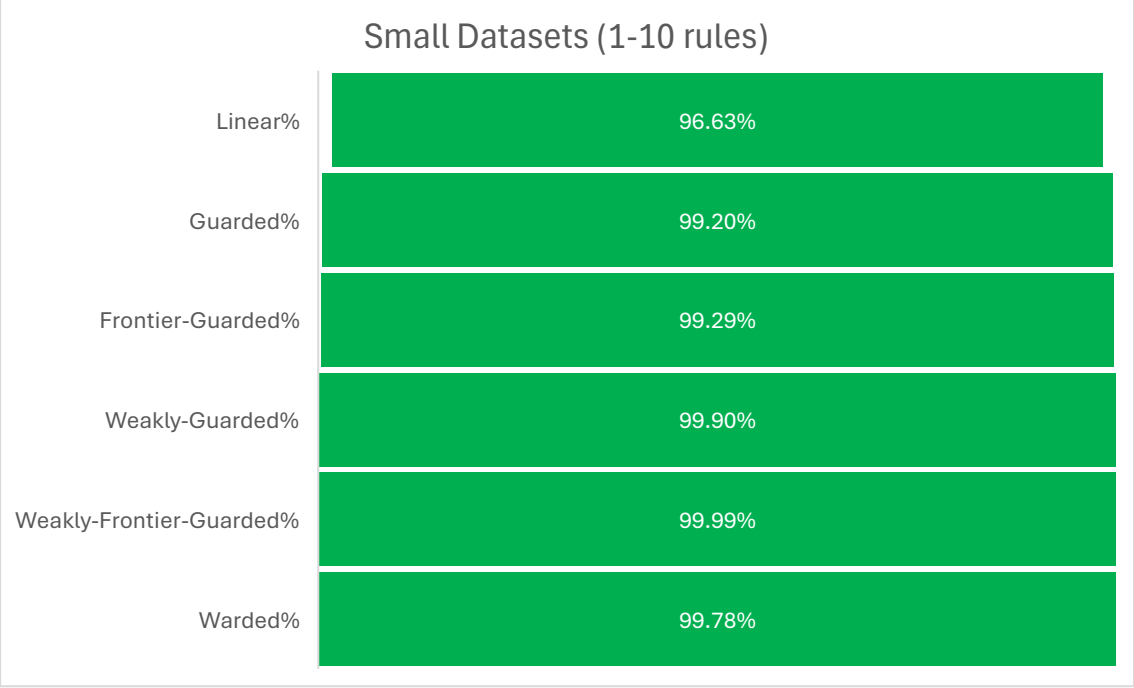


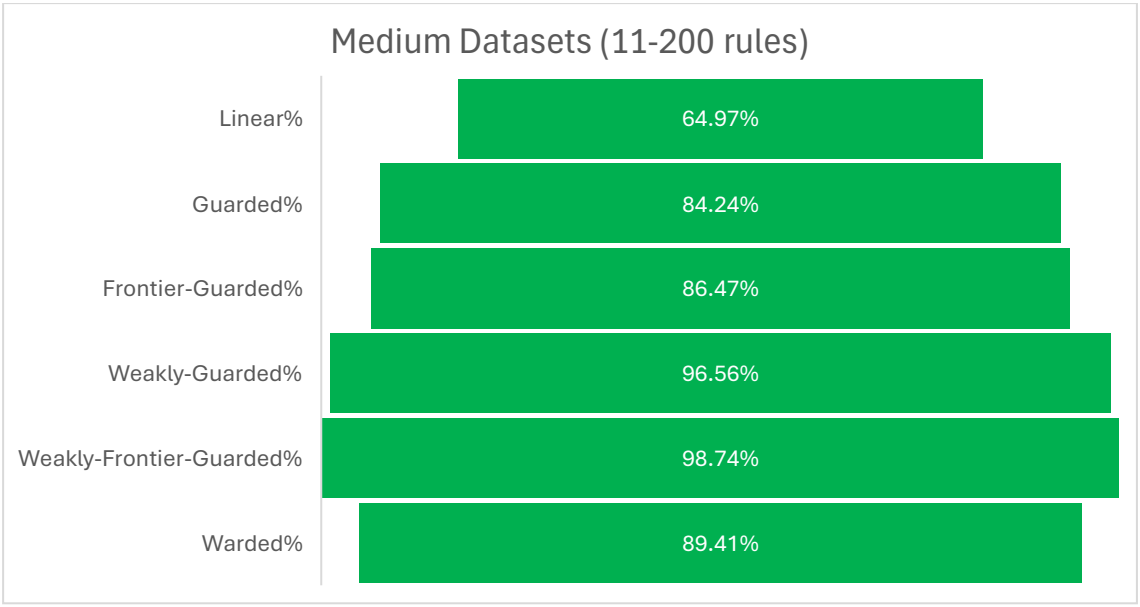

Percentage of datasets that belong to at least 1 language of the Guarded Family
Guarded Family
Outside

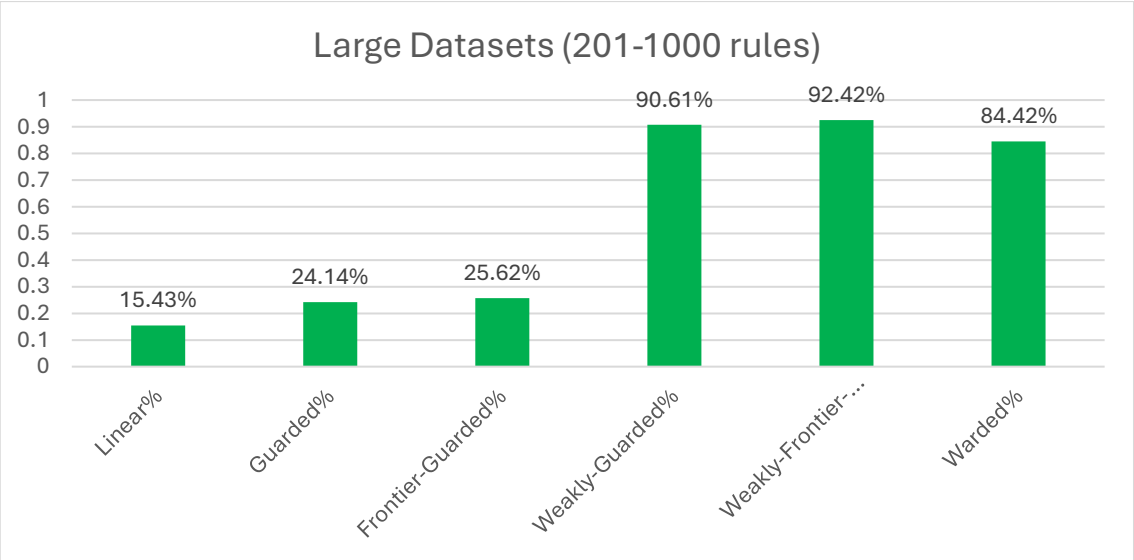

Weakly-Frontier-Guarded datasets
Also belong to Warded language
Do not belong to Warded languiage


Percentage of languages that belong to all Guarded Family languages
Belong to all Guarded Family languaes
Do not belong to all Guarded Family languaes


Number of datasets by size category
8000
7000
6000
5000
4000
3000
2000
1000
0
6907
5401
3985
2258
Small
Medium
Large
Very Large


Small Datasets (1-10 rules)
Linear%
96.63%
Guarded%
99.20%
Frontier-Guarded%
99.29%
Weakly-Guarded%
99.90%
Weakly-Frontier-Guarded%
99.99%
Warded%
99.78%


Medium Datasets (11-200 rules)
Linear%
64.97%
Guarded%
84.24%
Frontier-Guarded%
86.47%
Weakly-Guarded%
96.56%
Weakly-Frontier-Guarded%
98.74%
Warded%
89.41%



4.5.8 Analysis based on dataset size

In order to further investigate the behavior of real datasets, an additional analysis was performed based on the size of the datasets.

As introduces earlier, the datasets were categorized into 4 groups according to their size and those 4 categories are:

Small datasets (1–10 rules)

Medium datasets (11–200 rules)
Large datasets (200–1000 rules)
Very Large datasets (1000+ rules)
This categorization a deeper investigation and understanding of how the structural complexity of datasets differs based of their size and how this affects the Guarded Family languages that also differ from one category to another.

Behavior in small datasets:
The result show that small datasets exhibit very high percentages at all Guarded Family languages and they are the 37% of the total datasets, the biggest category.

Almost all small datasets belong to every language of the Guarded Family, since the most restrictive one which is Linear has a 96% rate in this category. This suggests that for rulesets with a small number of rules:
-The structure of rules is simple
-Joins are minimal

This shows that small-scale datasets tend to naturally comply with all languages of the Guarded Family since they do not require complex structures.

Behavior in medium datasets:
In medium-size datasets, the percentages remain relatively high but a slight diversion between the languages is starting to happen. While the most expressive languages like Weakly-Frontier-Guarded still have a very high percentage other most strict languages like Linear and Guarded show a noticeable decrease, especially the Linear language.
This demonstrates that for medium-size rulesets:
-As the number of the rules increases a bit, the joins become more frequent
-There are more dependencies between rules
-The stricter languages start to fall off

However, the differences between the languages are not extreme and medium datasets maintain a relatively balanced structure.

Behavior in large datasets:

The most interesting behavior in our opinion appears in large datasets, where there is a clear separation between the 3 expressive languages and the 3 simple languages.

The simpler languages (Linear, Guarded, Frontier-Guarded) show much lower rates compared to before as they range from 15-25%.

On the other hand ,the more expressive languages (Warded, Weakly-Guarded, Weakly-Frontier-Guarded) still maintain high percentages as they range from 84-92%.

So, in this category there is a huge gap between the 2 categories. This phenomenon can be explained because:

-Large datasets contain complex joins and multiple dependencies

-The interaction and dependencies between rules increase

-Null propagation is more complex

As a result, strict languages that do not support complex rules and propagation are no longer sufficient to model datasets of this size, while languages based on semantic properties are still efficient.

Behavior in very large datasets

Another very interesting category is the very large datasets, where a significant drop is observed across all languages. In these datasets, even the 3 expressive languages have a significant drop, their percentage was 84-92% in large datasets but here in very large datasets it drops to 35-47%, while the simpler languages maintain their low percentages ranging from 20-30% and unexpectedly they have a small increase in relation to their rates in large datasets.

Here the huge gap that existed between the 2 categories disappears and their percentages are relatively close. This indicates that:

-Extremely large datasets introduce very complex interactions

-The propagation of null values becomes highly dynamic

-Even advanced constrains cannot fully operate for such sizes

Observations: Structural complexity vs language expressiveness

Another important finding of our thesis is that dataset size is strongly correlated with languages percentages. More specifically:

Small datasets → Almost perfect for all languages

Medium datasets → Balanced behavior

Large datasets → Require expressive languages

Very large datasets → Challenge for all languages

So as the dataset size increases, the rules complexity increases, the null propagations increases and the need for more expressive languages becomes essential.

Warded in different dataset sizes

Because the first finding of the dataset is that warded is the most balanced language, we want to comment on its behavior in the dataset categories as well. The behavior of the Warded language across datasets sizes further strengthens the main conclusion of this wok.

It is clear that it consistently maintains:

-Very high percentages for small and medium datasets

-Strong performance in large datasets

-Its rates are always close to the rates of Weakly-Guarded and Weakly-Frontier-Guarded, but not always close to Frontier-Guarded, Guarded and Linear

So, since it avoids the extreme complexity of Weakly-Guarded and Weakly-Frontier-Guarded and it maintains high percentages close to them for alla categories while the other simple languages do not do that, it confirms that Warded:

-Adapts well to increase of dataset size

-Controls null propagation even for more complex rules effectively

-Is locally stable across different dataset sizes not just globally balanced

So, it is in fact the best and most practical language.

Overall conclusion of size-based analysis

This analysis based on datasets size highlights something important for languages analysis that is not visible in aggregate statistics alone. While global percentages show that most datasets belong to a lot of Guarded Family languages, this analysis shows that dataset complexity plays a critical role in language classification, simple languages may have relatively high rates but they are only useful for small datasets, expressive languages do not just have very high rates but they are necessary for large-scale datasets and Warded keeps being consistent and balanced across all scales.

4.6 Discussion and Interpretation of Results

The experimental evaluation presented in Section 4.5 provides a picture of the behavior of real rulesets regarding Guarded Family languages. The results confirm the theoretical properties of the languages and reveal important features of real data which cannot be concluded immediately from theory. In this section, a comprehensive interpretation of the results aims to:

-Connect them with the theoretical concepts presented in Chapter 2

-Draw general conclusions about the behavior of the datasets

-Evaluate the practical significance of the languages

4.6.1 Confirmation of the theoretical hierarchy

An expected observation from the results is that the percentages of datasets that belong to each language increase as we move from less expressive to more expressive languages.

Specifically, it is observed that Linear presents the lowest percentage, followed by the Guarded percentage, the Weakly-Frontier-Guarded has the highest percentage and there is no direct relationship between the Warded, Frontier-Guarded and Weakly-Guarded percentages as none is a subset of any other, but based on the strictness of their definition it is concluded that the Weakly-Guarded is the most expressive and the Frontier-Guarded the least expressive.

This agreement of the results with theory is important, as it functions as an indication of the correctness of the implementation and the system, without it there would be a strong indication of an error in the system.

4.6.2 Behavior of real datasets

One of the most significant findings of our research is that real datasets show a clear preference for more expressive languages.

That is because more strict languages like Linear and Guarded do not cover a large percentage of datasets while more expressive languages cover the majority of cases. This shows that real datasets contain multiple joins and complex dependencies.

This finding is particularly important as it shows that datasets from real application are not always simple or constrained but incorporate with complex logical relationships. At the same time, the fact that expressive languages cover most of them suggests that they are not completely chaotic and in practice there is an implicit structure and discipline in the way they are constructed.

4.6.3 The role of null propagation

One of the main theoretical issues analyzed in Chapter 2 is the propagation of null values though existential variables, affected positions and dangerous variables.

Experimental results confirm that the simple syntactic structure languages like Linear are not sufficient to control complexity while languages based on semantic properties (affected positions, dangerous variables) are closer to the behavior of real data.

This is clearly seen especially on large datasets where there are complex rules and multiple joins, where the different between percentage of datasets that belong to Linear/Guarded/Frontier-Guarded and Warded/Weakly-Guarded/Weakly-Frontier-Guarded. This confirms that the problem is not only the structure of the rules but the propagation of information within the ruleset and that is why languages that directly aim at controlling this propagation are the most practical.

4.6.4 Syntactic vs Semantic restrictions

Another important observation which comes from the experiments is the distinction between syntactic and semantic restriction in Guarded Family languages.

Linear, Guarded and Frontier-Guarded are based on syntactic properties of rules like number of atoms in the body or guard atom for all variables. On the other hand, Weakly-Guarded, Warded and Weakly-Frontier-Guarded rely on semantic properties like affected positions, dangerous variables etc. that are directly related to the null propagation. The experiment results show this categorization strongly and it seems that in all dataset sizes the 3 languages of each category move in proportions close to the other languages of their category and that semantic restrictions provide a more realistic and effective way to model and analyze real datasets.

4.6.5 Limitations of aggregate analysis

Although the overall percentages provide a clear overview of the distribution of datasets across the Guarded Family, they only show the big picture and not the underlying data. They hide important variations between datasets of different sizes and we cannot know if languages perform good for every size using only the big picture.

This limitation motivated the additional analysis based on dataset sizes presented in Section 4.5.8, where more detailed and interesting results came out showing that dataset sizes play a crucial role in the applicability of each language. Therefore, global statistics are useful for general observations, but a deeper analysis is needed to fully understand the behavior of each language.

4.6.6 Practical implications

The findings of this work have important implications for the practical use of rule-based languages. In real world applications, a language must be able to effectively model datasets while having acceptable computational cost, the results show that in general semantic languages are the most useful and Warded is the best language to use for balance

but if someone is only interested in small datasets it might be best to use a simpler language. It is important to design languages that strike a balance between complexity and control for them to be useful in every situation, but depending on the situation everyone can draw conclusions from the results about which language is the most appropriate to do the analysis they want.

Chapter 5

Conclusions

This thesis focused on the study of Guarded Family languages and on the experimental investigation of their presence in real rulesets. These languages are subsets of Tuple-Generating Dependencies, and their aim is to extract information through logical rules while keeping a balance between expressiveness and computational cost. Although their theoretical properties have been studied their practical use in real datasets has not been investigated to the same extend and this work attempted to do that by developing an analysis system and performing experiments on real data.

Initially, the theoretical background of TGDs was presented and all the languages of the Guarded Family were analyzed with emphasis on the concepts of affected positions and variable classification, with the way each language uses them to limit the propagation of null values.

Then, an analyzer was developed in Java for the automatic recognition of the Guarded Family languages at ruleset level. The implementation included a rule parser, atom and variable representations, positions and variables analysis mechanisms and specific checks for each language. At the same time, a preprocessing pipeline and experiment automation with Python scripts were implemented.

The experimental results of the work led to particularly important conclusions. The first main conclusion is that all Guarded family languages show relatively high percentages in real datasets that satisfy their requirements. This shows that the limitation of guarded-based languages are not just theoretical constructs but reflect real structure and can be implement in real-world rulesets and ontologies with efficiency.

The most important finding of the work concerns the Warded language. The results showed that approximately 85% of the datasets belong to the Warded language. This is an important result as it shows that Warded achieves an excellent balance between

expressiveness and computation efficiency because it is expressive enough to cover the majority of real datasets and it maintains better computation properties compared to Weakly-Guarded and Weakly-Frontier-Guarded languages, so it's the most practical language to use in order to model real datasets.

Another important conclusion emerged from the analysis of the datasets based on their size. It was observed that for small datasets all languages show very high rates over 90% but in large datasets there is a clear separation between the simplest and the expressive languages as Linear, Guarded and Frontier-Guarded have percentages less than 26% while Warded, Weakly-Guarded and Weakly-Frontier-Guarded have percentages over 84%. This result shows that as the number of rules and the complexity of the dataset increases, the need for expressive languages that can handle complex interactions becomes more necessary. At the same time, it shows that Warded continues to maintain high rates close to Weakly-Guarded and Weakly-Frontier-Guarded and far ahead of the other languages which strengthens the conclusion that it is the most balanced and best practical language.

Although the work produced significant results, there is considerable potential for future expansion.

Initially, the study was limited to the Guarded Family languages. A natural continuation would be to extend the analyzer to other important TGD families such as Sticky, Weakly-Sticky or any other languages to do a larger comparative study between different decidable fragments.

Another important direction concerns the optimization of the analyzer's performance and the possible use of parallel processing. The analyzer was not designed with the aim of the best performance, it has structures used for performance and reuse of results that make it quite efficient but could be improved even more especially with parallel programming.

In addition, something important in the optimization of the analyzer that would significantly reduce execution time would be to stop checking language checkers if we see that a ruleset belongs to a language that is a subset of another. For example, if a ruleset

belongs to the Linear language, then it could automatically stop checking for all other languages and assume that it belongs to all of them. Another example if it does not belong to Linear TGDs but belongs to Gaured TGDs then it could assume that it also belongs to Frontier-Guarded, Weakly-Guarded and Weakly-Frontier-Guarded languages and check only the Warded language checker. This will save a lot of calculations and time in the execution process.

Finally, it would be interesting to connect static ruleset analysis with real query answering systems, so out system can automatically recognize the languages to which a ruleset belongs to and then select the most appropriate reasoning and optimization techniques for efficient query execution.

Overall, the work shows that Guarded Family languages and especially Warded language are important tools for knowledge modeling and logical reasoning. The results confirm that it is possible to have a balance between expressiveness and efficiency with languages that restrict the propagation of null values, which makes these languages suitable for real-world applications of knowledge representation and ontology reasoning.

Bibliography

[1] J.-F. Baget, M.-L. Mugnier, S. Rudolph, and M. Thomazo, “Walking the Complexity Lines for Generalized Guarded Existential Rules” in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 2011, pp. 712–717. [Online]. Available: https://www.ijcai.org/Proceedings/11/Papers/126.pdf

[2] G. Berger, G. Gottlob, A. Pieris, and E. Sallinger, “The Space-Efficient Core of Vadalog” *ACM Transactions on Database Systems*, vol. 47, no. 1, Article 1, pp. 1–46, 2022. [Online]. Available: https://scispace.com/pdf/the-space-efficient-core-of-vadalog-1q9oyskn.pdf

[3] P. Bourhis, M. Manna, M. Morak, and A. Pieris, “Guarded-Based Disjunctive Tuple-Generating Dependencies” *ACM Transactions on Database Systems*, vol. 41, no. 4, Article 27, 2016. [Online]. Available: https://www.pure.ed.ac.uk/ws/files/28293747/TODS_16_2.pdf

[4] G. Gottlob, M. Manna, M. Morak, and A. Pieris, “On the Complexity of Ontological Reasoning under Disjunctive Existential Rules” in *Proceedings of the International Workshop on Description Logics*, 2012. [Online]. Available: https://www.mat.unical.it/datalog-exists/pub/12mfcs.pdf