Bottom-Up Evaluation of Second-Order Datalog with Negation

Antonis Gkanios
AL1.19.003

Supervisor:
Angelos Charalambidis, Asst. Prof., Dept. of Informatics and Telematics, Harokopio University of Athens
Co-supervisor:
Panagiotis Rondogiannis, Prof, Dept. of Informatics and Telecommunications, National and Kapodistrian University of Athens.

ABSTRACT

Extending logic programming beyond the first-order paradigm has long been a challenge due to the complexity of handling higher-order constructs and negation. This thesis investigates the evaluation of second-order Datalog with negation, which extends the expressiveness of traditional logic programming while maintaining declarative semantics. Prior research introduced a three-valued immediate consequence operator to compute the well-founded model for higher-order programs, yet the computational overhead of this approach limits its scalability.

We propose a bottom-up evaluation framework that combines program transformation techniques with alternating fixpoint evaluation to systematically refine approximations of the well-founded model. This approach introduces a new way to handle negation in second-order logic programs and presents an alternative framework for evaluating the well-founded semantics. It contributes to a deeper understanding of higher-order logic programming and its potential use in complex reasoning tasks.

ΣΥΝΟΨΗ

Η επέκταση του λογικού προγραμματισμού πέρα από τη λογική πρώτης τάξης αποτελεί εδώ και καιρό μια πρόκληση λόγω της πολυπλοκότητας του χειρισμού δομών υψηλότερης τάξης αλλά και της άρνησης. Η παρούσα εργασία διερευνά την αποτίμηση προγραμμάτων Datalog δεύτερης τάξης με άρνηση, η οποία επεκτείνει την εκφραστικότητα του παραδοσιακού λογικού προγραμματισμού διατηρώντας παράλληλα τη δηλωτική του σημασιολογία. Προηγούμενη έρευνα εισήγαγε μεθόδους για τον υπολογισμό well-founded μοντέλων σε προγράμματα υψηλότερης τάξης, ωστόσο η υπολογιστική επιβάρυνση αυτής της προσέγγισης περιορίζει την άμεση επεκτασιμότητα της

Στα πλαίσια αυτής της εργασίας, προτείνουμε μία bottom-up προσέγγιση που συνδυάζει τεχνικές μετασχηματισμού προγράμματος με μεθόδους εύρεσης σταθερού σημείου ώστε να βελτιώσει συστηματικά τις προσεγγίσεις του well-founded μοντέλου ενός προγράμματος. Αυτή η προσέγγιση εισάγει έναν νέο τρόπο χειρισμού της άρνησης σε λογικά προγράμματα δεύτερης τάξης και παρουσιάζει ένα ολοκληρωμένο πλαίσιο για την αποτίμηση τους. Τέλος, συμβάλλει στη βαθύτερη κατανόηση του λογικού προγραμματισμού υψηλότερης τάξης και της πιθανής χρήσης του σε σύνθετες συλλογιστικές εργασίες.

CONTENTS

1 Introduction **1**
- 1.1 Related Work . . . 2

2 Second-Order Logic Programs **5**

3 Evaluation of First-Order Datalog Programs **7**
- 3.1 Relational model for first-order datalog programs . . . 7
  - 3.1.1 Operations on Relations . . . 9
- 3.2 Evaluation for positive programs . . . 10
  - 3.2.1 Pattern Matching . . . 10
  - 3.2.2 Generate Variable Substitutions in Literals . . . 11
  - 3.2.3 Combining Relations . . . 12
  - 3.2.4 Propagating Variable Substitutions to Head Arguments . . . 12
  - 3.2.5 Evaluation . . . 13
- 3.3 Evaluation for programs with negation . . . 14
  - 3.3.1 Double Program . . . 15
  - 3.3.2 Evaluating Programs . . . 16

4 Evaluation of Second-Order Programs **19**
- 4.1 Relational model for second-order Datalog programs . . . 19
- 4.2 Evaluation for positive programs . . . 21
  - 4.2.1 Pattern Matching . . . 21
  - 4.2.2 Generate Variable Substitutions in Literals . . . 22
  - 4.2.3 Combining Relations . . . 24
  - 4.2.4 Adapting VTOA and EVAL Algorithms . . . 25
- 4.3 Evaluation for programs with negation . . . 28
  - 4.3.1 Double Program . . . 28
  - 4.3.2 Pattern Matching in Second-Order Programs . . . 29
  - 4.3.3 Propagating Variable Substitutions to Head Arguments . . . 32
  - 4.3.4 Complement of a Relation in the Second-Order Case . . . 33
  - 4.3.5 Combining Relations . . . 37
  - 4.3.6 Adapting VTOA and EVAL Algorithms . . . 37
- 4.4 Example Evaluations . . . 39

5 Conclusion and Future work **45**

Bibliography **49**

CHAPTER 1 INTRODUCTION

Declarative programming paradigms are mainly represented by two approaches: functional programming and logic programming. While functional programming is widely recognized for its expressive use of higher-order functions, logic programming has traditionally operated at a first-order level. Attempts to extend logic programming into a higher-order form have historically encountered various challenges, limiting its general applicability. Despite limited success in specialized domains like theorem proving and meta-programming, achieving a comprehensive, general-purpose higher-order logic programming language has remained elusive.

The development of extensional higher-order logic programming, as presented in [1], marked a significant milestone. This advancement introduced a general-purpose programming language that maintains the foundational properties of classical first-order logic while incorporating higher-order constructs. Specifically, it enabled the use of predicates as arguments and allowed for predicate variables, making the language expressive and suitable for a wide range of applications.

The extensional, three-valued semantics for higher-order logic programs with negation, introduced in [4], provides an immediate consequence operator for determining the well-founded model (WFM) of a program. However, despite being designed for this purpose, it faces challenges related to efficiency of the computation. The operator iteratively adjusts two approximations: an under-approximation that progressively increases and an over-approximation that gradually decreases until they converge. While operations on both approximations can be computationally expensive, this is more easily observed in the case of the over-approximation, due to its typically larger initial size. To illustrate, consider the following second-order logic program:

```
non_subset(P, Q) ← P(X), not Q(X).
subset(P, Q) ← not non_subset(P, Q).
```

Now, let us define a set of constants $C = \{c_1, \ldots, c_n\}$. The over-approximation is initialized in a way that every predicate in the program is considered to be true for every possible combination of its arguments. In this example we would have to generate every unary set P, Q such that $P \subseteq C$ and $Q \subseteq C$. Generating all these sets is computationally very expensive as C grows in size. In the general case, if C contains n elements and a predicate has k arguments, where each argument is a predicate symbol of arity ρ_i, then the total combinations that we will need to generate will be $\prod_{i=1}^{i=k} 2^{\rho_i n}$

In this thesis, we focus on the case of second-order Datalog with negation, specifically the second-order, function-free subset of the language introduced in [4]. We present a bottom-up evaluation method that combines the concept of program transformation, inspired by Kemp et al. [5], with the alternating fixpoint evaluation developed by van Gelder [6]. Our approach systematically addresses the complexities of handling negation, providing a more efficient solution for evaluating WFM of second-order Datalog programs.

The rest of the thesis is organized as follows:

- Chapter 2 introduces the syntax of the language used in this thesis. It covers the key constructs and formal rules that define expressions within our logic framework.
- Chapter 3 studies the basic algorithms required for evaluating first-order programs, covering both positive programs and programs with negation.
- Chapter 4 explores the evaluation of second-order programs. It details the necessary program transformations, adjustments to algorithms, and techniques for handling negation in a higher-order setting.
- Chapter 5 concludes the thesis by summarizing the key findings, highlighting the contributions of our evaluation methods, and suggesting directions for future research.

1.1 Related Work

The study of second-order Datalog with negation has been influenced by various reasoning frameworks. One key approach is the use of chase-based techniques, which have been adapted to second-order settings to facilitate reasoning over complex dependencies [9]. The chase algorithm has traditionally been used in database theory to ensure logical consistency by iteratively applying existential rules. Recent work has extended this method to incorporate set-based reasoning, improving its applicability for handling second-order constructs. By utilizing existential rules in conjunction with chase procedures, researchers have developed approaches that systematically derive logical conclusions while ensuring termination under specific conditions.

Another significant development is the application of Answer Set Programming (ASP) techniques to second-order logic [10]. ASP provides a nonmonotonic reasoning framework that aligns well with second-order Datalog, particularly for handling negation. The introduction of ASP with sets has optimized grounding strategies, reducing computational overhead while maintaining expressive power. This extension has enabled more efficient evaluations of second-order logic programs, particularly when dealing with complex rule dependencies. However, challenges remain in efficiently managing predicate variable instantiation and optimizing interactions between higher-order rules and negation.

The evaluation of higher-order logic programs has also been explored in the context of HiLog, a language that extends first-order Datalog with higher-order syntactic capabilities while maintaining a first-order semantics [11]. Research on HiLog has investigated bottom-up evaluation methods, adapting traditional naive and seminaive evaluation techniques for higher-order constructs. The work demonstrates that established Datalog evaluation algorithms can be effectively extended to HiLog, supporting recursive queries and complex dependencies. These techniques contribute to a broader understanding of how higher-order logic programs can be efficiently evaluated in a bottom-up manner.

Despite these prior contributions, existing approaches do not fully address the specific challenges posed by second-order Datalog with negation. Chase-based techniques rely on stratified programs to handle negation, which imposes restrictions on expressiveness. Our approach does not require stratification, allowing for a more general handling of negation in second-order logic. ASP-based methods rely on extensive grounding, leading to unnecessary computations. Our approach eliminates excessive grounding while utilizing well-founded semantics instead of stable model semantics.

The methodology introduced in this thesis aims to bridge these gaps by combining bottom-up evaluation with program transformation techniques tailored specifically for second-order Datalog with negation. By refining well-founded semantics and utilizing structured transformations, this approach ensures a more efficient evaluation process that reduces redundant computations and minimizes unnecessary derivations. Future research can build upon this work by further optimizing transformation rules, integrating hybrid reasoning techniques, and applying these methods to real-world domains where second-order reasoning is essential.

CHAPTER 2

SECOND-ORDER LOGIC PROGRAMS

In this chapter we introduce Second-Order Datalog with Negation ($\mathcal{SODN}$). Our language is based on a simple type system that supports two base types: o, the boolean domain, and ι, the domain of individuals (data objects).There are two classes of composite types: predicate π (assigned to predicate symbols) and argument ρ (assigned to parameters of predicates).
A type can either be predicate, or argument, denoted by π and ρ respectively.

Definition 2.1. Predicate and argument types are defined to be:

$$\pi := (\rho_1, \dots, \rho_n) \to o$$
$$\rho := \iota \mid (\iota_1, \dots, \iota_n) \to o$$

Definition 2.2. The alphabet of the $\mathcal{SODN}$ consists of the following:

- Predicate variables of every predicate type π (denoted by capital letters such as P and Q).
- Predicate constants of every predicate type π (denoted by lowercase letters such as p and q).
- Individual variables of type ι (denoted by capital letters such as X and Y).
- Individual constants of type ι (denoted by lowercase letters such as a and b).
- The following logical constant symbols are defined: the conjunction symbol $\wedge$, the inverse implication symbol $\leftarrow$, and the negation operator $\neg$.
- Left "(" and right ")" parentheses.

Definition 2.3. The set of terms is defined as follows:

- Every predicate variable (respectively predicate constant) of type π is a term of type π;
- Every individual variable (respectively individual constant) of type ι is a term of type ι;

- If $E_1, \dots, E_n$ are terms of type $\rho_1, \dots, \rho_n$ respectively and E is a term of type $(\rho_1, \dots, \rho_n) \to o$ then $E(E_1, \dots, E_n)$ is a term of type o.

Definition 2.4. The set of expressions of $\mathcal{SODN}$ is defined as follows:

- A term of type ρ is an expression of type ρ.
- If E is an expression of type o then $\neg E$ is an expression of type o.

To denote that an expression E has type τ we will write $E : \tau$.

Definition 2.5. An atom is an expression of type o of the form $p(E_1, \dots, E_n)$ where each E_i is either a variable or a constant. p is referred to as the predicate of the atom denoted as $pred(p(E_1, \dots, E_n))$. The variables of the atom will be expressed as $vars(p(E_1, \dots, E_n))$.

Definition 2.6. A literal is either an atom or the negation of an atom. If q is an atom then q is a positive literal and $\neg q$ is a negative literal.

Definition 2.7. A rule of $\mathcal{SODN}$ is of the form $p(V_1, \dots, V_n) \leftarrow L_1 \wedge \cdots \wedge L_m$ where p is a predicate constant of type $(\rho_1, \dots, \rho_n) \to o$, $\{V_i\}_{1 \le i \le n}$ are distinct arguments of type ρ_i respectively. $\{L_i\}_{1 \le i \le m}$ are literals.

The term $p(V_1, \dots, V_n)$ will be referred as the head of the rule and the conjunction $L_1 \wedge \cdots \wedge L_m$ as the body of the rule. A rule is considered safe if every variable in the head of the rule also appears in at least one literal in the body of the rule. From this point forward, we will only consider rules that are safe.

Definition 2.8. A program of $\mathcal{SODN}$ is a finite set of rules.

We will borrow some features of Prolog's syntax while writing examples and when presenting the algorithms for the evaluation. Specifically, instead of the conjunction symbol $\wedge$, we will use commas to separate each expression in the body and end it with a full stop. Finally we will use 'not' keyword instead of the negation constant operator $\neg$.

Example 2.9. *Tthe following is a legitimate second-order program*

```
closure(R,X,Y) ← R(X,Y).
closure(R,X,Y) ← R(X,Z), closure(R,Z,Y).
```

Definition 2.10. The set of predicates of a program $\mathcal{P}$, denoted as $pred(\mathcal{P})$, is defined as follows:

$$pred(\mathcal{P}) = \{p,\ L_1, \dots, L_n \mid\ p(V_1, \dots, V_n) \leftarrow L_1(a_1, \dots, a_m), \dots, L_n(c_1, \dots, c_k) \in \mathcal{P}\ \}$$

Definition 2.11. The Herbrand universe $\mathcal{U}_{\mathcal{P}}$ of a program $\mathcal{P}$ is the set of all data constants that appear in the program.

CHAPTER 3
EVALUATION OF FIRST-ORDER DATALOG PROGRAMS

In this section, the goal is to familiarize the reader with the bottom-up evaluation process that is described in [8].

We begin by defining the general notion of a relation, which serves as the foundational data structure for representing facts and intermediate results in Datalog programs. Relations encapsulate sets of tuples, and their properties directly influence how predicates are internally represented and manipulated during evaluation. Building upon this foundation, we will explore how the predicates of a Datalog program are stored internally.

The next step is the bottom-up evaluation of positive first-order Datalog programs. By first-order, we mean that predicate variables or predicate constants are not allowed as arguments in literals. Here, we focus on algorithms for computing their model, as they do not involve negation, making evaluation simpler. For cases with negation, we introduce the double program transformation to compute the well-founded model, providing a consistent way to handle negation.

Through this chapter, we aim to build a clear understanding of the algorithms and transformations necessary for evaluating first-order Datalog programs. Starting with positive programs and extending to those with negation, this exploration establishes a comprehensive foundation for the evaluation of more complex logic programming paradigms.

3.1 Relational model for first-order datalog programs

In this part, we will define the relational model for our language based on well-established mathematical concepts of relations, using the model defined in [7]. This approach provides a structured framework for organizing and evaluating our logic programs with negation.

The term *relation* is used here in its well-established mathematical sense. Given a collection of sets $S_1, S_2, \ldots, S_n$ (which may or may not be distinct), a relation $\mathcal{R}$ is defined as a subset of their Cartesian product:

$$\mathcal{R} \subseteq S_1 \times S_2 \times \cdots \times S_n.$$

The Cartesian product of sets $S_1, \ldots, S_n$ is the set of all n-tuples $(v_1, \ldots, v_n)$ such that $v_i \in S_i$ for every $1 \leq i \leq n$. For example, if we have $n = 2$, $S_1 = \{0, 1\}$, $S_2 = \{a, b, c\}$ then $S_1 \times S_2 = \{(0, a), (0, b), (0, c), (1, a), (1, b), (1, c)\}$.

A relation is defined as any subset of this Cartesian product. For example, a relation might consist of a smaller subset of tuples, such as $\{(0, a), (1, a), (1, b)\}$. The empty set is another example of a relation. Relations are finite in most database contexts.

The arity or degree of a relation corresponds to the number of domains involved. For example, a relation involving two domains has an arity of 2, and its elements are pairs. A relation involving three domains has an arity of 3, and its elements are triplets. Each tuple contains exactly one value from each associated domain, and the number of components in a tuple matches the relation's arity.

A relation can also be viewed as a table, where each row corresponds to a tuple in the relation, and each column represents one component of the tuple. The columns are given descriptive names $N_1, N_2, \ldots,$ referred to as attributes, which collectively define the relation scheme. For a relation $\mathcal{R}$ with attributes $(N_1, \ldots, N_k)$, $scheme(\mathcal{R}) = \{N_1, \ldots, N_k\}$. To get access to a specific column N_i we use the notation $\mathcal{R}[N_i]$. Viewing a relation in tabular form makes it easier to visualize the data and precisely define operations, such as selection, projection, and join, by leveraging the clear organization of rows and columns.

For the example above, if we associate name N_i with set S_i then this is how our relation would look like:

N_1	N_2
0	a
1	a
1	b

Since we are currently focusing on first-order Datalog, every set S_i can only contain individual values as its members. These values are atomic and cannot be higher-order constructs such as sets, sets of sets, or other complex structures.

In Datalog, the underlying mathematical model is the relational model, where predicate symbols represent relations. For each predicate symbol p in a program $\mathcal{P}$, we define a relation $rel(p)$. The collection of those relations is called the database of the program, denoted as $DB(\mathcal{P})$. Tuples stored in these relations are exactly those that make the predicate true. Each relation is represented as a set of ordered lists, with the components of a tuple appearing in a fixed order. The scheme of the relation is determined by the type of the predicate, where columns correspond to the arguments of the predicate and are referenced solely by their position. For example, if p is a predicate symbol, then $p(X, Y, Z)$ refers to a relation where X, Y, and Z represent the first, second, and third components, respectively, of some tuple in $rel(p)$. Each column in these relations is associated with values from the $\mathcal{U}_{\mathcal{P}}$.

To facilitate working with these relations, we can assign generic or "dummy" names, such as $C_1, C_2, C_3, \ldots$, to the columns. These names act as placeholders corresponding to the position of the components in the tuple, making it easier to reference and interpret the data.

Example 3.1. *If we consider the following program $\mathcal{P}$:*

```
p(a,b).
p(b,c).
q(a,b,c).
```

```
k(X) ← p(X,b), q(X,b,c).
```

then we would store three distinct relations, one for each predicate: p*,* q*, and* k*. Each relation is created with an arity that matches the number of arguments in the corresponding type of the predicate. These relations will be used to represent the facts defined by* p *and* q*, as well as the derived results for* k *based on the program's rules. This is how the relations will look after the evaluation of* $\mathcal{P}$ *is completed:*

C_1	C_2
a	b
b	c

Table 3.1: $rel(p)$

C_1	C_2	C_3
a	b	c

Table 3.2: $rel(q)$

C_1
a

Table 3.3: $rel(k)$

The details of the evaluation process will be discussed in the next section.

3.1.1 Operations on Relations

In this section, we define two fundamental operations on relations: the *natural join* and the *union*. These operations are essential for manipulating relations in various computations and play a crucial role in the evaluation of logic programs.

Natural Join

The *natural join*, denoted as ($\bowtie$), is an operation that combines two relations by merging tuples that agree on all shared attributes. Formally, the natural join of two relations $\mathcal{R}_1 \subseteq S_1 \times S_2 \times \cdots \times S_m$ and $\mathcal{R}_2 \subseteq T_1 \times T_2 \times \cdots \times T_n$ is defined as:

$$\mathcal{R}_1 \bowtie \mathcal{R}_2 = \left\{ (s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n) \;\middle|\; \begin{array}{l} (s_1, s_2, \ldots, s_m) \in \mathcal{R}_1, \\ (t_1, t_2, \ldots, t_n) \in \mathcal{R}_2, \\ s_k = t_j \text{ for all } k, j \text{ where } c_k = c_j \end{array} \right\}$$

Here, c_k and c_j represent the shared column names in scheme($\mathcal{R}_1$) and scheme($\mathcal{R}_2$), respectively. The resulting relation contains tuples that agree on all shared attributes.

Union

The *union* of two relations, denoted as ($\cup$), combines all tuples from both relations without duplication. Formally, the union of two relations $\mathcal{R}_1 \subseteq S_1 \times S_2 \times \cdots \times S_m$ and $\mathcal{R}_2 \subseteq S_1 \times S_2 \times \cdots \times S_m$ (with the same schema) is defined as:

$$\mathcal{R}_1 \cup \mathcal{R}_2 = \left\{ t \;\middle|\; t \in \mathcal{R}_1 \text{ or } t \in \mathcal{R}_2 \right\}.$$

This operation requires that both relations share the same schema (i.e., the same number and type of attributes). The resulting relation contains all distinct tuples from $\mathcal{R}_1$ and $\mathcal{R}_2$.

3.2 Evaluation for positive programs

In this section, we examine the evaluation process of Datalog programs, focusing on the concept of the *immediate consequence operator* T_P. This operator applies the rules of a Datalog program to existing facts to derive new facts according to the heads of those rules. Starting with an empty relation for each predicate, the operator iteratively derives new information by repeatedly applying the program's rules.

Formally, let $\mathcal{P}$ be a Datalog program and I an interpretation (a set of facts for the program predicates). The immediate consequence operator T_P is defined as:

$$T_P(I) = \bigcup_{\substack{r \in \mathcal{P} \\ r : h \leftarrow L_1, \ldots, L_n}} \{h \mid h \text{ is derived from } r \text{ under } I\},$$

where $r : h \leftarrow L_1, \ldots, L_n$ represents a rule in P, and I provides the facts that have been derived in previous steps for evaluating the body literals $L_1, \ldots, L_n$.

For recursive Datalog programs, T_P can be repeatedly applied on the facts derived by its previous applications. This iterative process continues until no new facts can be produced. It is straightforward to see that T_P is a monotone operator, meaning that if $I_1 \subseteq I_2$, then $T_P(I_1) \subseteq T_P(I_2)$.

Through the systematic application of T_P, we compute the result of a Datalog program, capturing all derivable conclusions from the rules and facts. This section provides a detailed exploration of this operator and its role in the bottom-up evaluation process.

3.2.1 Pattern Matching

Pattern matching is a crucial step in the evaluation process. For a rule to infer information about the predicate in its head, each literal in the body must evaluate to true. The corresponding relations contain tuples that satisfy the predicates of these literals. If the argument tuple of a literal can be matched with a tuple in the relation of its predicate, the literal is considered true and satisfied.

Given a literal $p(t_1, \ldots t_n)$ and a tuple $(\mu_1, \ldots, \mu_n) \in rel(p)$, we are looking for a way to match every t_i with the corresponding μ_i. Since we know that $(\mu_1, \ldots, \mu_n)$ makes p true, a successful match means the literal will also be evaluated as true. If every t_i is a constant, then it is sufficient to check if $\forall i \; t_i = \mu_i$. In the case where there are variables among the literal's terms, the match is successful if there exists a substitution τ for the variables that would make the argument tuple identical to the tuple $(\mu_1, \ldots, \mu_n)$.

A substitution τ is formally defined as a partial mapping $\tau : \mathcal{V} \to \mathcal{C}$, where $\mathcal{V}$ is α set of variables and $\mathcal{C}$ is set of constants. For a variable $x \in \mathcal{V}$, if $\tau(x) = c$ for some $c \in \mathcal{C}$, then x is substituted by c.

The algorithm below finds and returns the substitution τ

Algorithm 1: First Order Matching Algorithm

```
Input  : $p(t_1, \dots, t_n)$
Input  : $(\mu_1, \dots, \mu_n)$: tuple $\in rel(p)$
Output: $\tau$ containing substitution for variables
$\tau \leftarrow \{X_i : \epsilon \mid X_i \in vars(p(t_1, \dots, t_n))\}$;
for $i \leftarrow 1$ to $n$ do
    if $t_i$ is variable then
        if $\tau(t_i) = \epsilon$ then
            $\tau(t_i) \leftarrow \mu_i$;
        end
        else if $\tau(t_i) \neq \mu_i$ then
            return $\emptyset$;
        end
    end
    else if $t_i$ is constant $\wedge$ $\tau(t_i) \neq \mu_i$ then
        return $\emptyset$;
    end
end
return $\tau$;
```

Example 3.2. *Given a literal of the form $p(X, Y, c)$ and a tuple (a, b, c) the algorithm above would produce substitution τ such that $\tau(X) = a, \tau(Y) = b$. If the given tuple was (a, b, b) then the algorithm would return $\emptyset$ as there would be a mismatch in the constant for the third argument.*

3.2.2 Generate Variable Substitutions in Literals

In the previous subsection, we explained that satisfying a literal $p(t_1, \dots, t_n)$ requires keeping track of potential substitutions for its variables. To structure this process, we introduce the *ATOV* algorithm. This algorithm relies on the matching algorithm described above to generate a relation where the attributes represent the variables in the literal's arguments. By applying the matching algorithm to the tuples in $rel(p)$, each substitution derived from a successful match is added as a tuple to this relation.

Algorithm 2: First Order ATOV

```
Input  : $p(t_1, \dots, t_n)$: literal with terms $t_1, \dots, t_n$
Input  : $rel(p)$: Relation for predicate $p$
Output: $\mathcal{R}$: output relation
$\mathcal{R} \leftarrow \emptyset$;
$scheme(\mathcal{R}) \leftarrow \{X_i \mid X_i \in vars(p(t_1, \dots, t_n))\}$;
foreach $(\mu_1, \dots, \mu_n) \in rel(p)$ do
    $\tau \leftarrow match(p(t_1, \dots, t_n), (\mu_1, \dots, \mu_n))$;
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{\tau(X_i) \mid X_i \in scheme(\mathcal{R})\}$;
end
return $\mathcal{R}$;
```

Example 3.3. *If for example we are examining the literal $p(X, Y, Z, d)$ against $rel(p)$ as it is presented below*

N_1	N_2	N_3	N_4
a	b	c	d
b	c	a	d
b	c	d	a
a	a	b	c
c	d	d	d

Table 3.4: $rel(p)$

we can see that only 1st, 2nd and 5th tuple are successfully matching with the literal's terms. The produced relation $\mathcal{R}$ will look like this:

X	Y	Z
a	b	c
b	c	a
c	d	d

Table 3.5: $\mathcal{R}$

3.2.3 Combining Relations

Since the body of a rule generally consists of multiple conjunctive literals, we need a method to ensure that the entire body is satisfied. This requires finding variable substitutions under which all literals in the body are simultaneously satisfied.

To achieve this, we use the *natural join* operation, as defined in the previous chapter, to combine relations corresponding to the literals in the body of the rule. By applying the natural join iteratively to these relations, we obtain a new relation whose tuples represent variable substitutions that satisfy the entire body of the rule.

This process ensures that the constraints imposed by each literal are respected simultaneously. The result is a relation that encapsulates all possible combinations of variable substitutions that make the rule's body true.

3.2.4 Propagating Variable Substitutions to Head Arguments

Up to this point, we have explained how to construct a relation that captures the conditions under which the body of a rule is satisfied. When the body is satisfied, we can infer that the argument tuple of the head satisfies the rule's predicate. To achieve this, we apply the substitution tuples from the relation of body variables to the argument tuple of the head, generating the argument tuples that satisfy the head of the rule. This process, referred to as the conversion from variables relation to head arguments (VTOA), is detailed in the algorithm below.

Algorithm 3: VTOA

```
Input  : Head of rule $h(t_1, \ldots, t_n)$
Input  : Relation $\mathcal{R}$
Output: Set of tuples that satisfy $h$
1 $new\_tuples \leftarrow \emptyset$;
2 foreach row $r \in \mathcal{R}$ do
3     $t \leftarrow \big(r[t_i]$ if $t_i \in$ vars$(h(t_1, \ldots, t_n))$ else $t_i$ for $t_i \in \{t_1, \ldots, t_n\}\big)$;
4     $new\_tuples \leftarrow new\_tuples \cup t$;
5 end
6 return new_tuples;
```

3.2.5 Evaluation

We have outlined three distinct steps for deriving correct inferences from a rule of the form $h : -L_1, \ldots, L_n$:

First, for each literal L_i in the rule's body, the ATOV algorithm is applied to generate the corresponding literal variables relation, denoted as $\mathcal{R}_i$.

Next, the natural join of these relations is computed, resulting in:

$$\mathcal{R} = \mathcal{R}_1 \bowtie \ldots \bowtie \mathcal{R}_n,$$

which captures the combined variable substitutions that satisfy all literals in the body.

Finally, the resulting relation $\mathcal{R}$ is used in the VTOA algorithm to produce a relation of tuples satisfying the head's predicate. These tuples can then be added to the corresponding relation for the predicate in the program.

Algorithm 4: Evaluate Rule

```
Input   : Rule $h : -L_1, \ldots, L_n$
Input   : $rel$: Set of relations
Output  : Tuples that satisfy $h$
1 foreach $L_i \in \{L_1, \ldots, L_n\}$ do
2     $\mathcal{R}_i \leftarrow ATOV(L_i, rel(L_i))$;
3 end
4 $\mathcal{R} \leftarrow \mathcal{R}_1 \bowtie \ldots \bowtie \mathcal{R}_n$;
5 return $VTOA(h, \mathcal{R})$
```

At the core of the Naive evaluation process lies the operation *EVAL*, which corresponds to the immediate consequence operator for first-order logic. This operation computes the relations for every predicate symbol appearing in a program $\mathcal{P}$ based on the current relations of the program's predicates. These relations represent the facts derived so far. The *EVAL* operation is defined as:

Algorithm 5: EVAL

```
Input  : Program $\mathcal{P}$
Input  : $rel$: Set of relations
Output : Relations for every predicate in $\mathcal{P}$

/* $r^h$ is the set of rules $\in \mathcal{P}$ having h as head           */
/* $t_i^h$ is the set of tuples produced by i-th rule of $r^h$        */
/* $t^h$ is the set of tuples produced by every rule of $r^h$       */
1 foreach distinct $h$ appearing in head of $\mathcal{P}$'s rules do
2     $i \leftarrow 0$;
3     foreach $r \in r^h$ do
4         $t_i^h \leftarrow$ evaluate_rule($r, rel$);
5         $i \leftarrow i + 1$;
6     end
7     $t^h \leftarrow \bigcup_{j=0}^{j=i} t_j^h$;
8 end
9 return $\bigcup_{h \in \mathcal{P}} t^h$
```

The naive evaluation begins with an empty relation for every predicate in the program. The *EVAL* operation is repeatedly applied to update these relations. This process continues until a fixpoint is reached, meaning no new tuples are inserted into any relation during the last application of *EVAL*. When the evaluation is completed, the tuples in the relation for each predicate are exactly those that make the predicate true. Under the closed-world assumption, any tuple not present in the relation is considered to make the predicate false.

3.3 Evaluation for programs with negation

When introducing negation into Datalog programs, we allow negative literals in the body of a rule. Unlike positive programs, where every program can be assigned a two-valued semantics (as discussed in the previous section), the presence of negation leads to situations where a two-valued semantics may not exist. This challenge arises due to the potential for logical cycles involving negation, which can create ambiguity in determining whether a predicate is true or false.

Consider the following example:

```
p ← not p.
```

In this program, the truth value of p cannot be determined under two-valued semantics because p depends negatively on itself. Assigning p the value true or false would lead to a contradiction.

To address such cases, the *well-founded semantics* provides a robust framework for defining the meaning of programs with negation. Well-founded semantics is a three-valued semantics in which, in addition to the traditional truth values of *true* and *false*, atoms can also be assigned the truth value *undefined*. This three-valued approach helps to resolve ambiguities by explicitly representing uncertainty when neither true nor false can be conclusively assigned.

In the following section, we present a bottom-up evaluation method for computing well-founded models of first-order programs with negation. This method is based on the alternating fixpoint approach described in [6], which provides a constructive framework for characterizing well-founded semantics. To effectively handle programs with negation, we introduce a program transformation technique called the *double program*, which serves as the foundation for applying the alternating fixpoint method.

3.3.1 Double Program

First, for every predicate symbol p appearing in the set of predicate symbols of the program $\mathcal{P}$, and for every literal of the form $p(t_1, \ldots, t_n)$ appearing in a negative context `not` $p(t_1, \ldots, t_n)$, we introduce a new predicate $p'(t_1, \ldots, t_n)$, ensuring that p' does not appear in $\mathcal{P}$. All occurrences of `not` $p(t_1, \ldots, t_n)$ in $\mathcal{P}$ are replaced with `not` $p'(t_1, \ldots, t_n)$. This gives the *original half* of the transformed program, denoted $\mathcal{P}_{\text{unprimed}}$.

Next, we create a duplicate of $\mathcal{P}$, where for every predicate symbol $p \in \mathcal{P}$, every occurrence of $p(t_1, \ldots, t_n)$ is replaced with $p'(t_1, \ldots, t_n)$, and every occurrence of `not` $p'(t_1, \ldots, t_n)$ is replaced with `not` $p(t_1, \ldots, t_n)$. This forms the *primed half* of the transformed program, denoted $\mathcal{P}_{\text{primed}}$.

The intuition behind this procedure is to compute the well-founded model of $\mathcal{P}$ by utilizing the two subprograms of the doubled program. One subprogram computes the true facts, while the other computes the complement of the false facts. Each subprogram is positive if the negated predicates are treated as fixed, allowing the fixpoint of each subprogram to be computed using standard bottom-up techniques for programs without negation. The following example demonstrates this approach.

The intuition behind this procedure is to compute the well-founded model of $\mathcal{P}$ by utilizing the two subprograms of the doubled program. One subprogram computes the true facts, while the other computes the complement of the false facts. Each subprogram is positive if the negated predicates are treated as fixed, allowing the fixpoint of each subprogram to be computed using standard bottom-up techniques for programs without negation. The following example demonstrates this approach.

Example 3.4. *Consider the following program $\mathcal{P}$*

```
t(a,a,b).
t(a,b,a).
p(X) ← t(X,Y,Z), not p(Y), not p(Z).
p(b) ←  not r(a).
```

The doubled program $D(\mathcal{P})$ is given by:

```
t(a,a,b).
t(a,b,a).
p(X) ← t(X,Y,Z), not p'(Y), not p'(Z).
p(b) ←  not r'(a).

t'(a,a,b).
```

```
t'(a,b,a).
p'(X) ← t'(X,Y,Z), not p(Y), not p(Z).
p'(b) ←  not r(a).
```

3.3.2 Evaluating Programs

Our goal in computing the well-founded model of a program is to determine two sets of relations: $\mathcal{T}$, representing the definitely true facts, and $\mathcal{F}$, representing the definitely false facts. Using the doubled program $D(\mathcal{P})$ we compute negative information in a complementary manner by deriving the complement of the definitely false facts ($\mathcal{U}$). The process incrementally computes the true facts and alternates with computing the complement of false facts. This alternation follows a sequence $\mathcal{T}_1, \mathcal{U}_2, \mathcal{T}_3, \dots$. Intuitively, the tuples in the relations corresponding to unprimed predicates represent facts that are considered definitely true. Conversely, the tuples in the relations corresponding to primed predicates represent facts that are not definitely false, meaning they are potentially true or undefined.

In order to achieve this, we need a slight modification to our *ATOV* algorithm, as presented in the previous section. When dealing with negative literals ($not\ p(t_1, \dots, t_n)$), we aim to generate variable substitutions that make the literal false. Specifically, we want every tuple $(t'_1, \dots, t'_n)$ such that $t_i \in \mathcal{U}_\mathcal{P}$ and $(t'_1, \dots, t'_n) \notin rel(p)$.

Algorithm 6: ATOV

```
    Input  : $p(t_1, \dots, t_n)$: literal with terms $t_1, \dots, t_n$
    Input  : $rel$: Set of relations for both primed and unprimed predicates
    Output: $\mathcal{R}$: output relation
1   $\mathcal{R} \leftarrow \emptyset$;
2   $scheme(\mathcal{R}) \leftarrow \{X_i \mid X_i \in vars(p(t_1, \dots, t_n))\}$;
3   $\Pi \leftarrow \begin{cases} \begin{cases} rel(p), & \text{if } p(t_1,\dots,t_n) \text{ is positive} \\ \mathcal{U}_P^n \setminus rel(p), & \text{if } p(t_1,\dots,t_n) \text{ is negative} \end{cases} & \text{if } p \text{ is unprimed,} \\ \begin{cases} rel(p'), & \text{if } p'(t_1,\dots,t_n) \text{ is positive} \\ \mathcal{U}_P^n \setminus rel(p'), & \text{if } p'(t_1,\dots,t_n) \text{ is negative} \end{cases} & \text{if } p \text{ is primed.} \end{cases}$
4   foreach $(\mu_1, \dots, \mu_n) \in \Pi$ do
5       $\tau \leftarrow match(p(t_1, \dots, t_n), (\mu_1, \dots, \mu_n))$;
6       $\mathcal{R} \leftarrow \mathcal{R} \cup \{\tau(X_i) \mid X_i \in scheme(\mathcal{R})\}$;
7   end
8   return $\mathcal{R}$;
```

The *EVAL* process remains the same for each half-program, with the only difference being that we use this updated version of the *ATOV* algorithm to handle negation. Additionally, we require relations for both the primed and unprimed predicates of the program.

Before defining the Naive evaluation procedure, it is important to clarify an assumption about primed literals. Primed literals are assumed to denote facts that are not definitely false. Consequently, in the first iteration, every negated literal of a primed predicate ($\neg p'(t_1, \dots, t_n)$) will fail.

This means that the initial computation of $\mathcal{T}$ (the set of definitely true facts) will only consider rules that do not contain any negative literals involving primed predicates.

Algorithm 7: Naive Evaluation with Negation

```
   Input  : $\mathcal{P}$: Program
   Output: $\mathcal{T}$: relations for definitely true facts
   Output: $\mathcal{U}$: relations for not definitely false facts
1  $\mathcal{P}_{unprimed}$ , $\mathcal{P}_{primed} \leftarrow D(\mathcal{P})$;
2  $\mathcal{P}^- \leftarrow \{r \mid$ rule in $\mathcal{P}_{unprimed}$ that does not contain negative literals$\}$;
3  $\mathcal{T} \leftarrow \{rel(p) = \emptyset \mid \forall$ predicate symbol $p \in \mathcal{P}\}$;
4  $\mathcal{U} \leftarrow \{rel(p') = \emptyset \mid \forall$ predicate symbol $p \in \mathcal{P}\}$;
5  $\mathcal{T} \leftarrow EVAL(\mathcal{P}^-, \mathcal{T}, \mathcal{U})$;
6  repeat
7      $\mathcal{U} \leftarrow EVAL(\mathcal{P}_{primed}, \mathcal{T} \cup \mathcal{U})$;
8      $\mathcal{T} \leftarrow EVAL(\mathcal{P}_{unprimed}, \mathcal{T} \cup \mathcal{U})$;
9  until $\mathcal{T}$ remains unchanged;
10 return $\mathcal{T}$,$\mathcal{U}$
```

Example 3.5. *We will now see in practice how the evaluation of the previous example works. By applying the Naive evaluation procedure, we will demonstrate the computation of the well-founded model step by step, alternating between the original and primed halves of the program.*

For this example, we divide the program into the following three sub-programs:

- $\mathcal{P}_{unprimed}$

```
t(a, a, b).
t(a, b, a).
p(X) ← t(X, Y, Z), not p'(Y), not p'(Z).
p(b) ← not r'(a).
```

- $\mathcal{P}_{primed}$

```
t'(a, a, b).
t'(a, b, a).
p'(X) ← t'(X, Y, Z), not p(Y), not p(Z).
p'(b) ← not r(a).
```

- $\mathcal{P}^-$

```
t(a, a, b).
t(a, b, a).
```

Initially $rel(p) = rel(t) = rel(r) = rel(p') = rel(t') = rel(r') = \emptyset$
After evaluating $\mathcal{P}^-$*, the initial set of relations* $\mathcal{T}$ *is given by:*

$$\mathcal{T} = \{rel(p),\ rel(r),\ rel(t)\}.$$

Individual relations are as follows:

$$rel(p) = \emptyset, \quad rel(r) = \emptyset, \quad rel(t) = \{(a, a, b), (a, b, a)\}.$$

Using the results of $\mathcal{P}^-$ as the basis, we proceed with the evaluation of the primed predicates ($\mathcal{P}_{primed}$) to compute their model. This results in:

$$\mathcal{U} = \{rel(t') = \{(a,a,b),(a,b,a)\},\ rel(p') = \{(a),(b)\}\}.$$

Next, using the updated primed relations, we recompute the model of the unprimed predicates ($\mathcal{P}_{unprimed}$). This gives:

$$\mathcal{T} = \{rel(t) = \{(a,a,b),(a,b,a)\},\ rel(p) = \{(b)\},\ rel(r) = \emptyset\}.$$

Finally, using the updated unprimed relations, we again compute the model of the primed predicates. This produces:

$$\mathcal{U} = \{rel(t') = \{(a,a,b),(a,b,a)\},\ rel(p') = \{(b)\},\ rel(r') = \emptyset\}.$$

At this point, a fixpoint is reached, as no further changes occur in the relations.

The interpretation of the results after the stabilization of the two sets of relations, $\mathcal{T}$ and $\mathcal{U}$, is as follows:
Given a predicate p and a tuple $(t_1,\dots,t_n)$:

1. If $(t_1,\dots,t_n) \in rel(p)$, then p is **true** for $(t_1,\dots,t_n)$.
2. If $(t_1,\dots,t_n) \notin rel(p)$ and $(t_1,\dots,t_n) \in rel(p')$, then p is **undefined** for $(t_1,\dots,t_n)$.
3. If $(t_1,\dots,t_n) \notin rel(p)$ and $(t_1,\dots,t_n) \notin rel(p')$, then p is **false** for $(t_1,\dots,t_n)$.

Here, $(t_1,\dots,t_n) \in \text{rel}(p)$ is shorthand for the statement that the tuple $(t_1,\dots,t_n)$ matches (using the matching algorithm described earlier) with a tuple from rel(p).

This classification ensures a consistent interpretation of the well-founded semantics, capturing the three possible truth values (true, false, undefined). Consistency here means that for every predicate p in the program $\mathcal{P}$, the relation $rel(p')$ is a superset of $rel(p)$, i.e., $rel(p') \supseteq rel(p)$.

CHAPTER 4
EVALUATION OF SECOND-ORDER PROGRAMS

In the previous sections, we have explored the relational model in the context of first-order logic, detailing how to define the model and compute it for both positive programs and programs involving negation. In this chapter, we extend this framework to second-order logic programs, adopting a similar structure to guide our exploration.

We begin by addressing the necessary changes to the relational model to accommodate higher-order constructs, such as relations that may contain. These changes form the foundation for supporting the expressive power of second-order logic within the relational model.

Next, we examine the evaluation of positive second-order programs. This involves outlining the modifications required in the algorithms to handle the additional complexity introduced by second-order constructs, while ensuring the evaluation remains logically sound.

Finally, we turn our attention to programs with negation in the second-order setting. Here, we adapt the techniques for handling negation in first-order programs to the second-order case, accounting for the interactions between higher-order constructs and negation. This section highlights the challenges and solutions involved in compuitng well-founded semantics of second-order logic programs.

Through this progression, we aim to provide a comprehensive framework for understanding and evaluating second-order programs, building upon the principles established for first-order logic while addressing the unique features and challenges of second-order logic.

4.1 Relational model for second-order Datalog programs

To extend the relational model defined for first-order Datalog programs to second-order Datalog programs, we need to accommodate the more complex types involved in second-order predicates. These types introduce higher-order constructs that require adjustments to the way relations are defined and represented.

Given a program $\mathcal{P}$ and a predicate $p \in pred(\mathcal{P})$ with type $(\rho_1, \dots, \rho_n) \to o$, we construct the corresponding relation $rel(p)$ to align with the structure imposed by this type. The key difference in the second-order setting lies in how we interpret and store the arguments of predicates, based on their types:

- If $\rho_i = \iota$: These arguments are handled in the same way as in the first-order case. Their values are taken directly from $\mathcal{U}_{\mathcal{P}}$, the domain of individuals. Each first-order argument is represented as an atomic value, with no additional structure required.
- If $\rho_i = (\iota_1, \ldots, \iota_n) \to o$, these arguments are handled as partial multi-valued functions[1]:

$$f : {\mathcal{U}_{\mathcal{P}}}^n \to \{0, 1\},$$

  where f is a partial mapping from the n-fold Cartesian product ${\mathcal{U}_{\mathcal{P}}}^n$ to the Boolean domain $\{0, 1\}$. In this representation, $f(t_1, \ldots t_n) = 1$ indicates that the tuple $(t_1, \ldots t_n) \in {\mathcal{U}_{\mathcal{P}}}^n$ is included in the set, while $f(t_1, \ldots t_n) = 0$ indicates that $(t_1, \ldots t_n)$ is excluded from the set. This representation is particularly useful because it can handle partial information, where some tuples may remain undefined (not mapped to either 0 or 1). This method provides a structured and adaptable way to interpret second-order arguments, ensuring that they are accurately represented as relations over individuals, consistent with the type of predicate.

Example 4.1. *Consider a predicate p with the type $(\iota, (\iota, \iota) \to o) \to o$. Here:*

- *The first argument of p, denoted as C_1, is a first-order argument of type ι (individual). Its values are taken directly from the domain $\mathcal{U}_{\mathcal{P}}$.*
- *The second argument of p, denoted as C_2, is a second-order argument of type $(\iota, \iota) \to o$, which represents a binary relation over individuals. For example:*

$$\{(a, b) \to 1, (a, c) \to 0, (b, c) \to 1\}.$$

This results in the following representation for $rel(p)$:

C_1	C_2
a	$\{(a, b) \to 1, (a, c) \to 0, (b, c) \to 1\}$
b	$\{(a, c) \to 1, (b, a) \to 0\}$
c	$\{(a, b) \to 0, (a, a) \to 0\}$

If we examine the first row of rel(p), we can interpret it as follows: the predicate p is true for every (C_1, C_2) such that $C_1 = a$, and C_2 is a set of pairs that satisfies the following conditions:

- $(a, b) \in C_2$, $(b, c) \in C_2$,
- $(a, c) \notin C_2$,
- *Any other pair not explicitly mentioned can either be included or excluded in C_2 without affecting the truth of p.*

The same logic applies to every row of rel(p), where the values of C_1 and the corresponding mappings in C_2 determine the truth conditions for p.

[1]A *multi-valued function* $f : A \mapsto B$ is like a function from A to B except that there may by more than one possible value $f(x)$ for a given $x \in A$

4.2 Evaluation for positive programs

In this section, we demonstrate how to transition from first-order programs to second-order programs by modifying existing algorithms and incorporating the extended relational model. These adjustments enable the evaluation of positive programs while inheriting the foundational principles of the first-order approach.

4.2.1 Pattern Matching

In the evaluation of first-order programs, matching is essential for determining the substitutions needed to satisfy literals. Although the process is relatively straightforward when dealing with literals containing only first-order arguments, additional complexities arise when predicate constants are introduced into the argument tuples of literals.

To build on the foundational concepts discussed earlier, we now extend our analysis to address cases where the argument tuples of literals may include predicates. This includes both predicate constants and predicate variables, which require distinct approaches to ensure correct matching and correctness of the evaluation process.

When predicate constants appear as arguments in a literal, such as $p(\dots, q, \dots)$, modifications to the matching process are required. Specifically, we must define the concept of "sets that match." This concept is based on the interpretation of sets stored in the database, where the stored sets act as minimal representatives, including only the necessary elements to describe the predicate.

Consider a literal $p(\dots, q, \dots)$, where q occupies the i-th position in the argument list. To match this literal with a tuple from $rel(p)$, we examine the i-th component of the tuple. In this case, the i-th position contains a set S of key-value pairs:

$$S = \{(k_1 \to v_1), (k_2 \to v_2), \dots, (k_m \to v_m)\},$$

where each k_j represents a key and v_j is the corresponding value. Since we examine positive programs, the rules of the program can only deduce positive information and thus $v_j = 1$ for all j.

The set S represents the minimal mappings required for the predicate. For a match to occur, the corresponding relation of the component in the argument tuple ($rel(q)$ in the example above) must be a superset of S in terms of sets of mappings. Formally, a set S' matches S if and only if:

$$S \subseteq S',$$

where the subset relation is interpreted in terms of mappings. This means that for each pair $(k_j \to v_j) \in S$, the key k_j must also exist in S'.

By enforcing this subset condition, the matching process ensures that the literal respects the semantics of predicate constants and the information stored in the database.

Algorithm 8: Second-Order Matching Algorithm

```
Input  : $p(t_1, \dots, t_n)$: predicate with arguments
Input  : $(\mu_1, \dots, \mu_n)$: tuple $\in rel(p)$
Output: $\tau$: substitution for variables, or $\emptyset$ if matching fails
$\tau \leftarrow \{X_i : \epsilon \mid X_i \in vars(p(t_1, \dots, t_n))\}$;
for $i \leftarrow 1$ to $n$ do
    if $t_i$ is a variable then
        if $\tau[t_i] = \epsilon$ then
            $\tau[t_i] \leftarrow \mu_i$;
        else
            if $t_i$ is a first-order variable and $\tau[t_i] \neq \mu_i$ then
                return $\emptyset$;
            end
            else if $t_i$ is a second-order variable then
                $\tau[t_i] \leftarrow \tau[t_i] \cup \mu_i$;
            end
        end
    end
    else if $t_i$ is a constant then
        if $t_i$ is a data-object and $t_i \neq \mu_i$ then
            return $\emptyset$;
        else if $t_i$ is a predicate constant then
            $\alpha \leftarrow rel(t_i)$;
            $\mu_i^1 \leftarrow \{k \mid k \to 1 \in \mu_i\}$;
            if $\mu_i^1 \nsubseteq \alpha$ then
                return $\emptyset$;
            end
        end
    end
end
return $\tau$;
```

4.2.2 Generate Variable Substitutions in Literals

The *ATOV* (Argument-to-Variable) algorithm, first introduced in the previous chapter, is a fundamental tool for constructing relations by associating argument tuples with variable substitutions. While the original ATOV algorithm is sufficient for first-order programs, it requires significant enhancements to handle the complexities of second-order cases. Specifically, literals of the form $R(t_1, \dots, t_n)$, where R is a predicate variable, introduce challenges that cannot be addressed using the first-order approach alone.

In the second-order case, the primary difference lies in how predicate variables are processed. Predicate variables are no longer treated as fixed relations; instead, they are interpreted as collections of mappings that define partial multi-valued functions. These mappings allow the algorithm to systematically assign meanings to predicate variables, capturing their role as dynamic higher-order constructs. This extension enables the ATOV algorithm to handle the additional expressiveness of second-order logic programs.

To enhance readability and understanding of the second-order ATOV process, we explicitly distinguish between two scenarios:

- **Constant predicates:** For literals involving constant predicates, the ATOV algorithm is largely identical to its first-order counterpart. The only difference is the incorporation of the new pattern matching algorithm introduced in the previous section. This updated matching mechanism ensures correctness while maintaining the simplicity of the original approach.
- **Predicate variables:** For literals involving predicate variables, the ATOV algorithm dynamically constructs relations by generating mappings based on the possible values of the literal's terms. Each predicate variable is interpreted as a collection of mappings representing partial multi-valued functions, enabling the algorithm to handle second-order cases effectively.

Since the constant predicate case remains unchanged from the first-order ATOV algorithm (apart from the updated pattern matching), we will focus on demonstrating the predicate-variable case below.

Algorithm 9: Second-Order ATOV

```
Input   : Literal $L(t_1, \dots, t_n)$.
Output  : Relation $\mathcal{R}$
/* L can either be constant or variable predicate        */
if L is a constant predicate then
    return Constant-Predicate-ATOV($L(t_1, \dots, t_n), rel(L)$);
end
else if L is a predicate variable then
    return Variable-Predicate-ATOV($L(t_1, \dots, t_n)$);
end
```

Algorithm 10: Variable-Predicate-ATOV

```
Input   : Literal $L(t_1, \dots, t_n)$.
Output  : Relation $\mathcal{R}$ containing possible substitutions of L's variables
$scheme(\mathcal{R}) \leftarrow \{X_i \mid X_i \in vars(L(t_1, \dots, t_n))\} \cup \{L\}$;
foreach $t_i \in (t_1, \dots, t_n)$ do
    $S_i \leftarrow \{t_i\}$ if $t_i$ is constant else $\mathcal{U_P}$ ;
end
$\mathcal{CP} \leftarrow S_1 \times \cdots \times S_n$;
foreach $(v_1, \dots, v_n) \in \mathcal{CP}$ do
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{v_1, \dots, v_n,\ (v_1, \dots, v_n) \rightarrow 1\}$;
end
return $\mathcal{R}$;
```

Example 4.2. *If we are examining the literal $L(X, Y, a)$, where L is a predicate variable, and the universe $\mathcal{U_P} = \{a, b, c\}$, the produced relation $\mathcal{R}$ will look like this:*

X	Y	L
a	a	$\{(a, a, a) \to 1\}$
a	b	$\{(a, b, a) \to 1\}$
a	c	$\{(a, c, a) \to 1\}$
b	a	$\{(b, a, a) \to 1\}$
b	b	$\{(b, b, a) \to 1\}$
b	c	$\{(b, c, a) \to 1\}$
c	a	$\{(c, a, a) \to 1\}$
c	b	$\{(c, b, a) \to 1\}$
c	c	$\{(c, c, a) \to 1\}$

Table 4.1: Relation $\mathcal{R}$ for the literal $L(X, Y, a)$

4.2.3 Combining Relations

As described earlier, in first-order programs, the body of a rule typically consists of multiple conjunctive literals. Ensuring that the entire body is satisfied involves finding variable substitutions under which all literals in the body are simultaneously satisfied. This is achieved using the *natural join* operation, which combines the relations corresponding to the literals in the body of the rule.

In the case of second-order programs, the logic for combining first-order variables remains the same: the natural join operation is used to handle them. However, for predicate variables (variables of type π), a different approach is necessary. These variables are treated not as simple placeholders for specific relations but as collections of characteristics describing a predicate. Instead of applying the natural join operation, we perform a *union* over the representative sets stored as values in the literal relation attributes of these variables. This captures all possible collections of characteristics associated with the predicate variable in the context of the body of a rule, ensuring that all combinations required to satisfy the literals in the rule's body are represented.

Given two relations $\mathcal{R}_1$ and $\mathcal{R}_2$, we describe a systematic method for computing the combined relation $\mathcal{CR}$, which integrates both first-order and second-order attributes. We begin by examining the columns of $\mathcal{R}_1$ and $\mathcal{R}_2$. If a column C with second-order attributes appears in both relations, it is renamed as $C_{\mathcal{R}_1}$ in $\mathcal{R}_1$ and $C_{\mathcal{R}_2}$ in $\mathcal{R}_2$ to prevent name conflicts. This renaming ensures that second-order attributes from both relations remain distinct in subsequent operations. Next, we perform a *natural join* on $\mathcal{R}_1$ and $\mathcal{R}_2$, focusing exclusively on the first-order columns. This operation retains only rows with matching values in the first-order columns of both relations. The resulting relation includes all first-order columns used in the join as well as the renamed second-order columns $C_{\mathcal{R}_1}$ and $C_{\mathcal{R}_2}$, preserving their distinct attributes. Once the natural join is complete, we process the second-order columns as follows: For each row in the resulting relation, consider the second-order columns $C_{\mathcal{R}_1}$ and $C_{\mathcal{R}_2}$. Compute the *union* of the representative sets stored in $C_{\mathcal{R}_1}$ and $C_{\mathcal{R}_2}$ for that row, formally defined as $C = C_{\mathcal{R}_1} \cup C_{\mathcal{R}_2}$. Replace $C_{\mathcal{R}_1}$ and $C_{\mathcal{R}_2}$ with the unified column C in the combined relation. The resulting relation $\mathcal{CR}$ includes first-order attributes obtained from the natural join and unified second-order attributes C, representing the union of second-order attributes from $\mathcal{R}_1$ and $\mathcal{R}_2$. This algorithm ensures that the combined relation accurately reflects both the shared first-order relationships and the complete set of second-order characteristics from the input relations.

Algorithm 11: Combine Relations

```
Input  : Relation $\mathcal{R}_1$
Input  : Relation $\mathcal{R}_2$
Output: Combined relation $\mathcal{CR}$
/* Let $\mathcal{F}$ and $\mathcal{S}$ be the columns that have first and
   second-order attributes respectively                                */
foreach $C \in (\mathcal{S} \cap scheme(\mathcal{R}_1) \cap scheme(\mathcal{R}_2))$ do
    Rename $C$ into $C_{\mathcal{R}_1}$ in $\mathcal{R}_1$;
    Rename $C$ into $C_{\mathcal{R}_2}$ in $\mathcal{R}_2$;
end
$\mathcal{CR} \leftarrow \mathcal{R}_1 \bowtie \mathcal{R}_2$;
foreach row $r \in \mathcal{CR}$ do
    $r' \leftarrow r$;
    foreach $C \in (\mathcal{S} \cap scheme(\mathcal{R}_1) \cap scheme(\mathcal{R}_2))$ do
        cs $\leftarrow r[C_{\mathcal{R}_1}] \cup r[C_{\mathcal{R}_2}]$;
        $r'[C] \leftarrow cs$;
    end
    Replace $r$ with $r'$ in $\mathcal{CR}$;
end
foreach $C \in (\mathcal{S} \cap scheme(\mathcal{R}_1) \cap scheme(\mathcal{R}_2))$ do
    Drop from $\mathcal{CR}$ columns $C_{\mathcal{R}_1}$ and $C_{\mathcal{R}_2}$ ;
end
return $\mathcal{CR}$;
```

Example 4.3. *Consider two relations $\mathcal{R}$ and $\mathcal{Q}$ as they are defined here:*

L	X	Y
$\{(a,b) \to 1\}$	a	b
$\{(b,c) \to 1\}$	b	c
$\{(a,c) \to 1\}$	a	c

Table 4.2: Relation $\mathcal{R}$

L	X	Z
$\{(a,a) \to 1\}$	a	a
$\{(b,b) \to 1\}$	a	b
$\{(c,c) \to 1\}$	a	c

Table 4.3: Relation $\mathcal{Q}$

Column L in $\mathcal{R}$ and $\mathcal{Q}$ will be renamed to $L_{\mathcal{R}}$ and and $L_{\mathcal{Q}}$ respectively.

X	Y	Z	$L_{\mathcal{R}}$	$L_{\mathcal{Q}}$
a	b	a	$\{(a,b) \to 1\}$	$\{(a,a) \to 1\}$
a	c	c	$\{(a,c) \to 1\}$	$\{(c,c) \to 1\}$

Table 4.4: Relation $\mathcal{CR}$ after performing natural join in first-order attributes

4.2.4 Adapting VTOA and EVAL Algorithms

The *VTOA* (Variable-to-Argument) and *EVAL* operations remain fundamentally the same as in the first-order case, serving their roles in applying substitutions and computing the

X	Y	Z	L
a	b	a	$\{(a,b) \to 1, (a,a) \to 1\}$
a	c	c	$\{(a,c) \to 1, (c,c) \to 1\}$

Table 4.5: Relation $\mathcal{CR}$ after performing union in second-order attributes

immediate consequence operator. In the second-order context, these operations rely on the updated algorithms for *ATOV*, matching, and combining relations to ensure the correct handling of second-order programs.

We will now present an example to illustrate the evaluation process in the second-order context.

Example 4.4. *Consider the following program $\mathcal{P}$ and $\mathcal{U}_{\mathcal{P}} = \{a, b, c\}$.*

```
start(a).
reach(G,Y)← G(Y, Z), start(Y).
reach(G,Y) ← G(X, Y), reach(G, X).
```

For the first rule, we have the following:

G	Y	Z
$\{(a,a) \to 1\}$	a	a
$\{(a,b) \to 1\}$	a	b
$\{(a,c) \to 1\}$	a	c
...	...	...
$\{(c,b) \to 1\}$	c	b
$\{(c,c) \to 1\}$	c	c

Table 4.6: Relation for $G(Y, Z)$

Y
a

Table 4.7: Relation for $start(Y)$

Combining these two relations, we get the following tuples for $rel(reach)$:

C_1	C_2
$\{(a,a) \to 1\}$	a
$\{(a,b) \to 1\}$	a
$\{(a,c) \to 1\}$	a

Table 4.8: $rel(reach)$

When evaluating the second rule we will have the following:

G	X
$\{(a,a) \to 1\}$	a
$\{(a,b) \to 1\}$	a
$\{(a,c) \to 1\}$	a

Table 4.9: Relation for $reach(G, X)$

G	X	Y
$\{(a,a) \to 1\}$	a	a
$\{(a,b) \to 1\}$	a	b
$\{(a,c) \to 1\}$	a	c
...	...	...
$\{(c,b) \to 1\}$	c	b
$\{(c,c) \to 1\}$	c	c

Table 4.10: Relation for $G(X, Y)$

After combining the relations we get the following for the body of second rule:

C_1	C_2
$\{(a,a) \to 1\}$	a
$\{(a,b) \to 1\}$	a
$\{(a,c) \to 1\}$	a
$\{(a,a) \to 1, (a,b) \to 1\}$	b
$\{(a,a) \to 1, (a,c) \to 1\}$	c
$\{(a,b) \to 1, (a,a) \to 1\}$	a
$\{(a,b) \to 1, (a,c) \to 1\}$	c
$\{(a,c) \to 1, (a,a) \to 1\}$	a

Table 4.11: $rel(reach)$

The bottom-up evaluation aims to construct all possible binary relations that include at least one tuple where the first element is a. This example represents a variation of the closure predicate, which defines reachability from point "a" in a graph, with the graph being represented as a relation of its edges.

4.3 Evaluation for programs with negation

The evaluation of second-order programs with negation introduces additional complexity due to the presence of negative literals such as $not\ p(R)$ or $not\ R(X, Y)$. These constructs require extending the existing algorithms for positive programs to accommodate negated predicate variables and their interactions within the framework of second-order logic.

To compute the well-founded model of second-order programs with negation, we must extend two key areas:

- **Adapting algorithms for well-founded semantics of first-order programs:** The existing algorithms must be adapted to handle second-order constructs. This involves managing predicate variables within negated literals to ensure consistency when extending the semantics to higher-order predicates.
- **Generalizing algorithms for positive second-order programs:** The evaluation of positive programs typically involves operations such as unions and joins. Incorporating negation requires additional steps to compute complementary sets and address undefined values. These generalizations ensure an accurate representation of the well-founded model, particularly when handling negated literals and second-order predicate variables.

The following examples illustrate the expressiveness of second-order programs with negation:

Example 4.5. *This program states that P is a subset of Q if there does not exist an X for which $P(X)$ is true while $Q(X)$ is false.*

```
non_subset(P,Q) ← P(X), not Q(X).
subset(P,Q) ← not non_subset(P,Q).
```

Example 4.6. *This program defines p as true for every R such that $a \in R$, $b \in R$, and $c \notin R$. Similarly, q is true for every R for which p is false.*

```
p(R) ← R(a), R(b), not R(c).
q(R) ← not p(R).
```

These examples highlight the need for algorithms capable of handling negation in second-order constructs. Building on the foundational algorithms for positive programs, we extend key processes such as *ATOV*, *matching*, and *combining relations* to interpret negative literals consistently and compute the well-founded model. The following sections will describe these adaptations in detail.

4.3.1 Double Program

We will again adopt the double program transformation introduced earlier in order to evaluate two different sets of relations ($\mathcal{T}$ and $\mathcal{U}$) denoting the definitely true and the non-definitely false facts, respectively. No changes are needed to the transformation itself for its application to second-order programs.

Example 4.7. *Consider the following program $\mathcal{P}$*

```
q(a).
p(R) ← R(b), not q(X), not R(c).
k(R) ← R(d), not p(R).
l(a) ← k(q).
```

The doubled program $D(\mathcal{P})$ is given by:

```
q(a).
p(R) ← R(b), not q'(X), not R(c).
k(R) ← R(d), not p'(R).
l(a) ← k(q).


q'(a).
p'(R) ← R(b), not q(X), not R(c).
k'(R) ← R(d), not p(R).
l'(a) ← k'(q).
```

4.3.2 Pattern Matching in Second-Order Programs

Pattern matching in second-order programs with negation extends the principles used in positive programs but introduces additional considerations for primed and unprimed literals. Specifically, when dealing with literals of the form $p(\dots, q, \dots)$, where q is a predicate symbol in the argument list, the matching process must respect the semantics of second-order constructs.

Given a literal $p(\dots, q, \dots)$, where q appears at the i-th position, the corresponding component in a tuple from $rel(p)$ is represented as a set of key-value pairs:

$$S = \{(k_1 \to v_1), (k_2 \to v_2), \dots, (k_m \to v_m)\},$$

where k_j represents a key, and v_j is the associated value (0 or 1). The interpretation of these key-value pairs depends on whether the literal p is primed or unprimed. For unprimed literals, the relation $rel(p)$ captures the definitely true facts for p. A successful match occurs if the following conditions hold for the i-th component S:

1. If $v_i = 1,$ then $k_i \in rel(q),$ ensuring q is true for k_i.
2. If $v_i = 0,$ then $k_i \notin rel(q'),$ ensuring q is false for k_i.

These conditions ensure consistency between the second-order argument q and the logical interpretation of the literal p.

Example 4.8. *Consider the literal $p(q)$ and a tuple $\tau = (\{(a) \to 1, (b) \to 1, (c) \to 0\})$ from $rel(p)$. Suppose the relations for q and q' are:*

$$rel(q) = \{a, b, c\}, \quad rel(q') = \{a, b, c, d\}.$$

We evaluate the conditions for each key-value pair in τ:

- *For $(a) \to 1$: Since $a \in rel(q)$, this condition is satisfied, as q is definitely true for a.*
- *For $(b) \to 1$: Similarly, $b \in rel(q)$, so this condition is also satisfied.*
- *For $(c) \to 0$: Here, c must not be in $rel(q')$, as q must be definitely false for c. However, $c \in rel(q')$, violating the condition.*

Since $c \in rel(q')$ contradicts the requirement that q be definitely false for c, the matching fails.

Primed literals relax the conditions used for unprimed literals, as they capture facts that make the predicate either true or undefined. The matching rules for in this case are as follows:

1. If $v_i = 1,$ then q must be at least undefined for k_i.
   This means k_i must either belong to $rel(q)$ (indicating q is true for k_i) or belong to $rel(q') \setminus rel(q)$ (indicating q is undefined for k_i).
2. If $v_i = 0,$ then q must be at most undefined for k_i.
   This means k_i must not belong to $rel(q')$ (ensuring q is false for k_i) or belong to $rel(q') \setminus rel(q)$ (indicating q is undefined for k_i).

These conditions broaden the scope of matching to include undefined facts, reflecting the semantics of primed relations in the well-founded model.

Example 4.9. *Consider the primed literal $p'(q)$ and a tuple $\tau = (\{(a) \to 1, (b) \to 1, (c) \to 0\})$ from $rel(p')$. Suppose the relations for q and q′ are:*

$$rel(q) = \{a, b\}, \quad rel(q') = \{a, b, c\}.$$

We evaluate the conditions for each key-value pair in τ:

- *For $(a) \to 1$: Since $a \in rel(q)$, this condition is satisfied, as q is definitely true for a.*
- *For $(b) \to 1$: Similarly, $b \in rel(q)$, so this condition is satisfied.*
- *For $(c) \to 0$: Here, $c \in rel(q')$ and $c \notin rel(q)$, satisfying the condition that q is undefined for c.*

Since all conditions are satisfied, τ successfully matches $p'(q)$.

By distinguishing between primed and unprimed literals, the pattern-matching process accommodates the nuances of second-order programs with negation, ensuring accurate evaluation in the well-founded model.These conditions form the foundation for matching literals in the second-order case, ensuring that the semantics of primed and unprimed literals are respected. The complete algorithm for pattern matching, incorporating these rules, is presented below.

Algorithm 12: Second-Order Matching Algorithm

```
Input  : $p(t_1, \ldots, t_n)$: predicate with arguments
Input  : $(\mu_1, \ldots, \mu_n)$: tuple $\in rel(p)$
Output: $\tau$: substitution for variables, or $\emptyset$ if matching fails
$\tau \leftarrow \{X_i : \epsilon \mid X_i \in \mathit{vars}(p(t_1, \ldots, t_n))\}$;
for $i \leftarrow 1$ to $n$ do
    if $t_i$ is a variable then
        if $\tau[t_i] = \epsilon$ then
            $\tau[t_i] \leftarrow \mu_i$;
        else
            if $t_i$ is a first-order variable and $\tau[t_i] \neq \mu_i$ then
                return $\emptyset$;
            end
            else if $t_i$ is a second-order variable then
                $\tau[t_i] \leftarrow \tau[t_i] \cup \mu_i$;
            end
        end
    end
    else if $t_i$ is a constant then
        if $t_i$ is a data-object and $t_i \neq \mu_i$ then
            return $\emptyset$;
        else if $t_i$ is a predicate constant then
            $\alpha^{\mathcal{T}} \leftarrow rel(t_i)$;
            $\alpha^{\mathcal{U}} \leftarrow rel(t_i')$;
            $\mu_i^1 \leftarrow \{k \mid k \rightarrow 1 \in \mu_i\}$;
            $\mu_i^0 \leftarrow \{k \mid k \rightarrow 0 \in \mu_i\}$;
            if $\mu_i^1 \cap \mu_i^0 \neq \emptyset$ then
                return $\emptyset$;
            end
            if $p$ is unprimed then
                if $\mu_i^1 \not\subseteq a^{\mathcal{T}}$ then
                    return $\emptyset$;
                end
                if $\mu_i^0 \cap a^{\mathcal{U}} \neq \emptyset$ then
                    return $\emptyset$;
                end
            end
            else if $p$ is primed then
                if $\mu_i^1 \not\subseteq a^{\mathcal{U}}$ then
                    return $\emptyset$;
                end
                if $\mu_i^0 \cap a^{\mathcal{U}} \neq \emptyset \ \wedge \mu_i^0 \cap (a^{\mathcal{U}} \setminus a^{\mathcal{T}}) \neq \emptyset$ then
                    return $\emptyset$;
                end
            end
        end
    end
end
return $\tau$;
```

4.3.3 Propagating Variable Substitutions to Head Arguments

The process of generating a relation where the attributes correspond to the variables in a literal's arguments must be modified to account for negation in second-order programs. As before, we distinguish between cases involving constant predicates and predicate variables in the *ATOV* operation.

For predicate variables, the handling of positive literals remains unchanged from the positive program case. Specifically, for a positive literal $R(t_1, \ldots, t_n)$, the resulting relation is constructed by considering all possible mappings that assign truth values to the literal's terms based on the current database state.

For negative literals of the form $not\ R(t_1, \ldots, t_n)$, the interpretation differs. Here, the relation represents every possible value of the literal's terms $(\mu_1, \ldots, \mu_n)$ that are not present in R. Formally, this is expressed as:

$$(\mu_1, \ldots, \mu_n) \to 0,$$

indicating that the tuple $(\mu_1, \ldots, \mu_n)$ is explicitly excluded from R, thereby satisfying the negation condition. More formally:

Algorithm 13: Variable-Predicate-ATOV

```
Input   : Literal $L(t_1, \ldots, t_n)$.
Output  : Relation $\mathcal{R}$ containing possible substitutions of $L$'s variables
$scheme(\mathcal{R}) \leftarrow \{X_i \mid X_i \in vars(L(t_1, \ldots, t_n))\} \cup \{L\}$;
foreach $t_i \in (t_1, \ldots t_n)$ do
    $S_i \leftarrow \{t_i\}$ if $t_i$ is constant else $\mathcal{U_P}$ ;
end
$\mathcal{CP} \leftarrow S_1 \times \cdots \times S_n$;
foreach $(v_1, \ldots, v_n) \in \mathcal{CP}$ do
    if $L$ is a positive literal then
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{v_1, \ldots, v_n,\ (v_1, \ldots, v_n) \to 1\}$;
    end
    else if $L$ is a negative literal then
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{v_1, \ldots, v_n,\ (v_1, \ldots, v_n) \to 0\}$;
    end
end
return $\mathcal{R}$;
```

Example 4.10. *If we are examining the literal* $not\ L(X, Y, a)$*, where L is a predicate variable, and the universe* $\mathcal{U_P} = \{a, b, c\}$*, the produced relation* $\mathcal{R}$ *will look like this:*

X	Y	L
a	a	$\{(a,a,a) \to 0\}$
a	b	$\{(a,b,a) \to 0\}$
a	c	$\{(a,c,a) \to 0\}$
b	a	$\{(b,a,a) \to 0\}$
b	b	$\{(b,b,a) \to 0\}$
b	c	$\{(b,c,a) \to 0\}$
c	a	$\{(c,a,a) \to 0\}$
c	b	$\{(c,b,a) \to 0\}$
c	c	$\{(c,c,a) \to 0\}$

Table 4.12: Relation $\mathcal{R}$ for the literal $not\ L(X, Y, a)$

4.3.4 Complement of a Relation in the Second-Order Case

Before introducing the final algorithm for handling constant-predicate ATOV, it is crucial to formalize the computation of the complement of a relation $\mathcal{R}$. This step is essential when determining substitutions for variables in negative literals (not $p(t_1, \dots, t_n)$).

In the first-order case, computing the complement was straightforward: given ${\mathcal{U}_{\mathcal{P}}}^n$, the Cartesian product of the domain, we subtracted all possible substitutions for $(t_1, \dots, t_n)$ that appear in $\mathcal{R}$. However, in the second-order case, this approach is insufficient because $\mathcal{R}$ includes sets of mappings, which must also be handled.

Let $S = \{k_1 \to v_1, k_2 \to v_2, \dots, k_n \to v_n\}$ be a set of mappings, where $v_i \in \{0, 1\}$ for all i. Define the complement of S as the set of all mappings where at least one value v_i is flipped.

For any subset $I \subseteq \{1, 2, \dots, n\}$, let S_I be the set of mappings obtained by flipping the values v_i for all $i \in I$. Formally:

$$S_I = \{k_1 \to v'_1, k_2 \to v'_2, \dots, k_n \to v'_n\},$$

where

$$v'_i = \begin{cases} 1 - v_i & \text{if } i \in I, \\ v_i & \text{if } i \notin I. \end{cases}$$

The set of all possible mappings T such that at least one value v_i is flipped is given by:

$$S' = \bigcup_{\substack{I \subseteq \{1,2,\dots,n\} \\ I \neq \emptyset}} S_I.$$

Here:

- $I \subseteq \{1, 2, \dots, n\}$ is a subset of indices that determines which values v_i are flipped.
- The condition $I \neq \emptyset$ ensures that at least one value is flipped.
- For any subset I, the flipping operation is defined element-wise, preserving the values of v_i for $i \notin I$.

Thus, S' represents the complement of S, capturing all configurations where at least one value v_i differs from the original set of mappings.

Example 4.11. *If $S = \{a \to 1, b \to 0, c \to 1\}$, then $\mathcal{F}$ would include:*

- *Flipping one value:*

$$\{a \to 0, b \to 0, c \to 1\}, \quad \{a \to 1, b \to 1, c \to 1\}, \quad \{a \to 1, b \to 0, c \to 0\}.$$

- *Flipping two values:*

$$\{a \to 0, b \to 1, c \to 1\}, \quad \{a \to 0, b \to 0, c \to 0\}, \quad \{a \to 1, b \to 1, c \to 0\}.$$

- *Flipping all three values:*

$$\{a \to 0, b \to 1, c \to 0\}.$$

This generalization ensures that we can handle second-order constructs where the complement operation must accurately account for configurations that differ from the original by flipping values. This framework is essential for processing substitutions in negative literals in the context of second-order programs.

Now that we have defined how to compute the complement of a relation, we can utilize this framework to process rows in a relation $\mathcal{R}$. In the relation $\mathcal{R}$, each row represents a conjunction of conditions that must hold.

For instance, consider a predicate $p(X_1, X_2, X_3)$ with arity 3. A row in $\mathcal{R}$, such as $(a, b, \{a \to 1, b \to 1\})$, implies the condition:

$$X_1 = a \land X_2 = b \land X_3 = \{a \to 1, b \to 1\}.$$

If $\mathcal{R}$ contains multiple rows, these rows indicate a disjunction of the conditions they represent. For example, two rows:

$$(a, b, \{a \to 1, b \to 1\}) \quad \text{and} \quad (a, c, \{a \to 1\}),$$

collectively represent the condition:

$$(X_1 = a \land X_2 = b \land X_3 = \{a \to 1, b \to 1\}) \lor (X_1 = a \land X_2 = c \land X_3 = \{a \to 1\}).$$

Satisfying either row is sufficient to satisfy the overall condition for $p(X_1, X_2, X_3)$.

To compute the complementary relation $\mathcal{R}'$, we negate the disjunction represented by the rows in $\mathcal{R}$. This negation transforms the condition into a conjunction of negated rows. Using the example above, the negation would be:

$$\neg((X_1 = a \land X_2 = b \land X_3 = \{a \to 1, b \to 1\}) \lor (X_1 = a \land X_2 = c \land X_3 = \{a \to 1\})).$$

Applying De Morgan's laws, this expands to:

$$(X_1 \neq a \lor X_2 \neq b \lor X_3 \neq \{a \to 1, b \to 1\}) \land (X_1 \neq a \lor X_2 \neq c \lor X_3 \neq \{a \to 1\}).$$

The resulting conjunction of negated rows represents the complementary relation $\mathcal{R}'$. To compute $\mathcal{R}'$ in practice, we iterate through each row of $\mathcal{R}$, negate the corresponding conditions, and then merge the results using the *Combine Relations* algorithm. This process ensures that $\mathcal{R}'$ correctly captures all configurations that do not satisfy the original relation $\mathcal{R}$.

Example 4.12. *Given the relation $\mathcal{R}$ as it is described above:*

X_1	X_2	X_3
a	b	$\{a \to 1, b \to 1\}$
a	c	$\{a \to 1\}$

Table 4.13: $\mathcal{R}$

To construct the complementary relation $\mathcal{R}'$, we identify all configurations that do not satisfy the original relation $\mathcal{R}$. This involves negating the conditions in each row of $\mathcal{R}$. The complement captures all possible assignments for X_1, X_2, and X_3 where at least one value or mapping does not match the original table. Specifically:

- For X_1 and X_2, any value from the Herbrand universe $\mathcal{U}_{\mathcal{P}} = \{a, b, c\}$ that differs from the given rows of $\mathcal{R}$ will appear in $\mathcal{R}'$.
- For X_3, any modification to the set of key-value mappings $\{a \to 1, b \to 1\}$ or $\{a \to 1\}$, such as flipping one or more values between 0 and 1, will be included.

X_1	X_2	X_3
b	b	$\{a \to 0, b \to 1\}$
b	b	$\{a \to 1, b \to 0\}$
b	b	$\{a \to 0, b \to 0\}$
b	c	$\{a \to 0, b \to 1\}$
b	c	$\{a \to 1, b \to 0\}$
b	c	$\{a \to 0, b \to 0\}$
c	b	$\{a \to 0, b \to 1\}$
c	b	$\{a \to 1, b \to 0\}$
c	b	$\{a \to 0, b \to 0\}$
c	c	$\{a \to 0, b \to 1\}$
c	c	$\{a \to 1, b \to 0\}$
c	c	$\{a \to 0, b \to 0\}$
b	a	$\{a \to 0\}$
c	a	$\{a \to 0\}$

Table 4.14: Complementary Relation $\mathcal{R}'$

This complementary relation includes all possible configurations that are not covered by $\mathcal{R}$. For example, the row $(b, b, \{a \to 0, b \to 1\})$ indicates a scenario where $X_1 = b$, $X_2 = b$, and X_3 includes a flipped mapping of $a \to 0$ compared to the original row in $\mathcal{R}$.

The complement $\mathcal{R}'$ ensures that all tuples violating any condition of $\mathcal{R}$ are explicitly represented. This comprehensive representation is essential for reasoning about negative literals and identifying substitutions in second-order programs.

Algorithm 14: Compute Complementary Relation

```
Input  : Relation $\mathcal{R}$
Input  : Herbrand universe $\mathcal{U_P}$
Output: Complementary relation $\mathcal{R}'$

1  $\mathcal{R}' \leftarrow \emptyset$;
2  $scheme(\mathcal{R}') \leftarrow scheme(\mathcal{R})$;
3  foreach row r in $\mathcal{R}$ do
4      foreach $X_i \in scheme(\mathcal{R})$ do
5          if $X_i$ is a first-order column then
6              $CV_{X_i} \leftarrow \mathcal{U_P} \setminus \{r[X_i]\}$;
7          end
8          else if $X_i$ is a second-order column then
9              foreach $S \subseteq r[X_i] \wedge S \neq \emptyset$ do
10                 $S' \leftarrow \{(key \rightarrow 1 - val) \mid (key \rightarrow val) \in S\}$;
11                 $CV_{X_i} \leftarrow CV_{X_i} \cup S'$;
12             end
13         end
14     end
15     $\mathcal{IR} \leftarrow_{X_i \in scheme(\mathcal{R})} (CV_{X_i})$;
16     $\mathcal{R}' \leftarrow$ Combine_Relations$(\mathcal{R}', \mathcal{IR})$ ;
17 end
18 return $\mathcal{R}'$;
```

With all the necessary concepts clarified, including handling negation, computing complementary relations, and managing sets of mappings, we are now ready to present the complete and final version of the constant-predicate ATOV algorithm. This version integrates these considerations to ensure accurate evaluation of literals involving constant predicates in second-order programs.

Algorithm 15: Constant-Predicate ATOV

```
Input  : $p(t_1, \ldots, t_n)$: literal with terms $t_1, \ldots, t_n$
Input  : $rel$: Set of relations for both primed and unprimed predicates
Input  : Herbrand universe $\mathcal{U_P}$
Output: $\mathcal{R}$: output relation
1  $\mathcal{R} \leftarrow \emptyset$;
2  $scheme(\mathcal{R}) \leftarrow \{X_i \mid X_i \in vars(p(t_1, \ldots, t_n))\}$;
3  foreach $(\mu_1, \ldots, \mu_n) \in rel(p)$ do
4      $\tau \leftarrow match(p(t_1, \ldots, t_n), (\mu_1, \ldots, \mu_n))$;
5      $\mathcal{R} \leftarrow \mathcal{R} \cup \{\tau(X_i) \mid X_i \in scheme(\mathcal{R})\}$;
6  end
7  if $p$ is positive literal then
8      return $\mathcal{R}$;
9  end
10 else if $p$ is negative literal then
11     return compute_complementary_relation$(\mathcal{R}, \mathcal{U_P})$;
12 end
```

4.3.5 Combining Relations

The *Combine Relations* method remains unchanged from its use in evaluating positive programs. However, in the second-order case, a key distinction arises: a second-order argument R can associate the same key $(t_1, \ldots, t_n)$ with both 0 and 1. While this may initially appear contradictory, it is a deliberate mechanism to represent relations that either cannot be matched against any other relation or can be matched against every possible relation. This duality provides a flexible and expressive framework for capturing various logical scenarios within the evaluation process.

For instance, the framework can represent relations that are inherently contradictory and cannot exist in any concrete evaluation. Simultaneously, it can also express relations that match every other relation, providing a way to generalize evaluation across all possible configurations.

Example 4.13. *Consider the following program $\mathcal{P}$:*

```
p(R) ← R(a), not R(a).
q(R) ← not p(R).
```

In this example, the evaluation produces:

$$rel(p) = \{\ \{(a) \to 1, (a) \to 0\}\ \}.$$

This means that p *is true for every relation R that simultaneously includes a and does not include a. Such a relation R cannot exist, as the conditions are logically contradictory.*

On the other hand, for q*, we have:*

$$rel(q) = \{\{(a) \to 1\}, \{(a) \to 0\}\}.$$

This means that q *is true for every relation R that either includes a or does not include a. Clearly, every relation satisfies this condition, as any relation must either contain or exclude a. This representation provides a formal way to express all possible relations.*

4.3.6 Adapting VTOA and EVAL Algorithms

The *VTOA* (Variable-To-Arguments) and *EVAL* algorithms remain fundamentally unchanged from their counterparts in the first-order case with negation. These algorithms continue to serve their respective roles in generating new tuples for the head predicates and iteratively updating the relations of the program.

The key distinction in the second-order case lies in the use of the modified versions of the *matching* and *ATOV* algorithms. These adaptations, as discussed earlier, account for the unique characteristics of second-order constructs, such as predicate variables, key-value mappings, and their interactions in both primed and unprimed contexts.

By leveraging these modified components, the *VTOA* and *EVAL* algorithms integrate seamlessly into the evaluation process, ensuring that the semantics of second-order programs with negation are accurately captured while maintaining the structural simplicity of the first-order framework.

4.4 Example Evaluations

Let us consider the following program $\mathcal{P}$, which defines the subset relation:

```
non_subset(P, Q) ← P(X), not Q(X).
subset(P, Q) ← not non_subset(P, Q).
```

This program specifies that P is a subset of Q if it is not the case that there exists an X for which $P(X)$ is true while $Q(X)$ is false. In other words, $P \subseteq Q$ holds when all elements satisfying $P(X)$ also satisfy $Q(X)$.

To demonstrate the evaluation of this program, we assume that the domain $\mathcal{U}_{\mathcal{P}}$ of $\mathcal{P}$ is $\{a, b, c\}$.

The double program for $\mathcal{P}$, which separates unprimed and primed predicates, is as follows:

```
non_subset(P, Q) ← P(X), not Q(X).
subset(P, Q) ← not non_subset'(P, Q).
```

```
non_subset'(P, Q) ← P(X), not Q(X).
subset'(P, Q) ← not non_subset(P, Q).
```

We start the evaluation with both $\mathcal{T}$ and $\mathcal{U}$ as empty for every predicate symbol.

non_subset	**subset**
$\emptyset$	$\emptyset$

$\mathcal{T}$

non_subset'	**subset**
$\emptyset$	$\emptyset$

$\mathcal{U}$

We begin the evaluation process by computing the model for $\mathcal{P}_{unprimed}$

For rule **(1)** of the program

```
non_subset(P,Q) ← P(X), not Q(X).
```

we will get the following

P	X
$\{a \rightarrow 1\}$	a
$\{b \rightarrow 1\}$	b
$\{c \rightarrow 1\}$	c

after evaluating `P(X)`

Q	X
$\{a \rightarrow 0\}$	a
$\{b \rightarrow 0\}$	b
$\{c \rightarrow 0\}$	c

after evaluating `not Q(X)`

By combining the information from both relations, we arrive at the combined relation presented here:

P	Q	X
$\{a \rightarrow 1\}$	$\{a \rightarrow 0\}$	a
$\{b \rightarrow 1\}$	$\{b \rightarrow 0\}$	b
$\{c \rightarrow 1\}$	$\{c \rightarrow 0\}$	c

after evaluating `body of rule (1)`

Regarding the second rule for the `subset` predicate, since $rel(non\_subset') = \emptyset$, the `get_false_combinations` algorithm will also return an empty relation. Consequently, no new tuples will be produced for the corresponding relation. As a result, $\mathcal{T}$ is as follows:

So $\mathcal{T}$ is the following:

non_subset	**subset**
$(\{a \rightarrow 1\}, \{a \rightarrow 0\})$	$\emptyset$
$(\{b \rightarrow 1\}, \{b \rightarrow 0\})$	
$(\{c \rightarrow 1\}, \{c \rightarrow 0\})$	

$\mathcal{T}$

As the next step, we are going to compute model $\mathcal{U}$ fo$\mathcal{P}_{unprimed}$. The first rule regarding `non_subset` predicate will be evaluated in the exact same way as in the evaluation of $\mathcal{P}_{unprimed}$. Therefore, we will only examine how the second rule will be evaluated.

```
subset'(P,Q) ← not non_subset(P,Q).
```

From previous computation of $\mathcal{T}$ we have the following for `non_subset`

P	Q
$\{a \rightarrow 1\}$	$\{a \rightarrow 0\}$
$\{b \rightarrow 1\}$	$\{b \rightarrow 0\}$
$\{c \rightarrow 1\}$	$\{c \rightarrow 0\}$

Intuitively, the relation $rel(non\_subset)$ contains every pair (P, Q) such that:

$$(P(a) = 1 \wedge Q(a) = 0) \vee (P(b) = 1 \wedge Q(b) = 0) \vee (P(c) = 1 \wedge Q(c) = 0)$$

To compute the complement of this relation, we apply De Morgan's law to the equation above. This gives us:

$$\neg\left((P(a) = 1 \wedge Q(a) = 0) \vee (P(b) = 1 \wedge Q(b) = 0) \vee (P(c) = 1 \wedge Q(c) = 0)\right),$$

which simplifies to:

$$(P(a) \neq 1 \vee Q(a) \neq 0) \wedge (P(b) \neq 1 \vee Q(b) \neq 0) \wedge (P(c) \neq 1 \vee Q(c) \neq 0).$$

This expression defines the conditions for pairs (P, Q) that are *not* in the corresponding relation for `non_subset`, effectively giving us its complement.

We will evaluate each part of the conjunction individually and then combine them using the previously defined **Combine Relations** algorithm. This process will yield the final result for the complement by conjuncting each evaluated component.

P	Q
$\{a \to 0\}$	$\{a \to 1\}$
$\{a \to 0\}$	$\{a \to 0\}$
$\{a \to 1\}$	$\{a \to 1\}$

$P(a) \neq 1 \vee Q(a) \neq 0$

P	Q
$\{b \to 0\}$	$\{b \to 1\}$
$\{b \to 0\}$	$\{b \to 0\}$
$\{b \to 1\}$	$\{b \to 1\}$

$P(b) \neq 1 \vee Q(b) \neq 0$

P	Q
$\{c \to 0\}$	$\{c \to 1\}$
$\{c \to 0\}$	$\{c \to 0\}$
$\{c \to 1\}$	$\{c \to 1\}$

$P(c) \neq 1 \vee Q(c) \neq 0$

After combining the intermediate relations the result will be the following:

P	Q
$\{a \to 0, b \to 0, c \to 0\}$	$\{a \to 0, b \to 0, c \to 0\}$
$\{a \to 0, b \to 0, c \to 0\}$	$\{a \to 1, b \to 0, c \to 0\}$
$\{a \to 0, b \to 0, c \to 0\}$	$\{a \to 0, b \to 1, c \to 0\}$
$\{a \to 0, b \to 0, c \to 0\}$	$\{a \to 0, b \to 0, c \to 1\}$
$\{a \to 0, b \to 0, c \to 0\}$	$\{a \to 1, b \to 1, c \to 0\}$
$\{a \to 0, b \to 0, c \to 0\}$	$\{a \to 1, b \to 0, c \to 1\}$
$\{a \to 0, b \to 0, c \to 0\}$	$\{a \to 0, b \to 1, c \to 1\}$
$\{a \to 0, b \to 0, c \to 0\}$	$\{a \to 1, b \to 1, c \to 1\}$
$\{a \to 1, b \to 0, c \to 0\}$	$\{a \to 1, b \to 0, c \to 0\}$
$\{a \to 1, b \to 0, c \to 0\}$	$\{a \to 1, b \to 1, c \to 0\}$
$\{a \to 1, b \to 0, c \to 0\}$	$\{a \to 1, b \to 0, c \to 1\}$
$\{a \to 1, b \to 0, c \to 0\}$	$\{a \to 1, b \to 1, c \to 1\}$
$\{a \to 0, b \to 1, c \to 0\}$	$\{a \to 0, b \to 1, c \to 0\}$
$\{a \to 0, b \to 1, c \to 0\}$	$\{a \to 1, b \to 1, c \to 0\}$
$\{a \to 0, b \to 1, c \to 0\}$	$\{a \to 0, b \to 1, c \to 1\}$
$\{a \to 0, b \to 1, c \to 0\}$	$\{a \to 1, b \to 1, c \to 1\}$
$\{a \to 0, b \to 0, c \to 1\}$	$\{a \to 0, b \to 0, c \to 1\}$
$\{a \to 0, b \to 0, c \to 1\}$	$\{a \to 1, b \to 0, c \to 1\}$
$\{a \to 0, b \to 0, c \to 1\}$	$\{a \to 0, b \to 1, c \to 1\}$
$\{a \to 0, b \to 0, c \to 1\}$	$\{a \to 1, b \to 1, c \to 1\}$
$\{a \to 1, b \to 1, c \to 0\}$	$\{a \to 1, b \to 1, c \to 0\}$
$\{a \to 1, b \to 1, c \to 0\}$	$\{a \to 1, b \to 1, c \to 1\}$
$\{a \to 1, b \to 0, c \to 1\}$	$\{a \to 1, b \to 0, c \to 1\}$
$\{a \to 1, b \to 0, c \to 1\}$	$\{a \to 1, b \to 1, c \to 1\}$
$\{a \to 0, b \to 1, c \to 1\}$	$\{a \to 0, b \to 1, c \to 1\}$
$\{a \to 0, b \to 1, c \to 1\}$	$\{a \to 1, b \to 1, c \to 1\}$
$\{a \to 1, b \to 1, c \to 1\}$	$\{a \to 1, b \to 1, c \to 1\}$

This is how $\mathcal{U}$ will look like:

subset	**non_subset**
$(\{a \to 0, b \to 0, c \to 0\}, \{a \to 1, b \to 0, c \to 0\})$	$(\{a \to 1\}, \{a \to 0\})$
$(\{a \to 0, b \to 0, c \to 0\}, \{a \to 0, b \to 1, c \to 0\})$	$(\{b \to 1\}, \{b \to 0\})$
$(\{a \to 0, b \to 0, c \to 0\}, \{a \to 0, b \to 0, c \to 1\})$	$(\{c \to 1\}, \{c \to 0\})$
$(\{a \to 0, b \to 0, c \to 0\}, \{a \to 1, b \to 1, c \to 0\})$	
$(\{a \to 0, b \to 0, c \to 0\}, \{a \to 1, b \to 0, c \to 1\})$	
$(\{a \to 0, b \to 0, c \to 0\}, \{a \to 0, b \to 1, c \to 1\})$	
$(\{a \to 0, b \to 0, c \to 0\}, \{a \to 0, b \to 0, c \to 0\})$	
$(\{a \to 1, b \to 0, c \to 0\}, \{a \to 1, b \to 0, c \to 0\})$	
$(\{a \to 1, b \to 0, c \to 0\}, \{a \to 1, b \to 1, c \to 0\})$	
$(\{a \to 1, b \to 0, c \to 0\}, \{a \to 1, b \to 0, c \to 1\})$	
$(\{a \to 1, b \to 0, c \to 0\}, \{a \to 1, b \to 1, c \to 1\})$	
$(\{a \to 0, b \to 1, c \to 0\}, \{a \to 0, b \to 1, c \to 0\})$	
$(\{a \to 0, b \to 1, c \to 0\}, \{a \to 1, b \to 1, c \to 0\})$	
$(\{a \to 0, b \to 1, c \to 0\}, \{a \to 0, b \to 1, c \to 1\})$	
$(\{a \to 0, b \to 1, c \to 0\}, \{a \to 1, b \to 1, c \to 1\})$	
$(\{a \to 0, b \to 0, c \to 1\}, \{a \to 0, b \to 0, c \to 1\})$	
$(\{a \to 0, b \to 0, c \to 1\}, \{a \to 1, b \to 0, c \to 1\})$	
$(\{a \to 0, b \to 0, c \to 1\}, \{a \to 0, b \to 1, c \to 1\})$	
$(\{a \to 0, b \to 0, c \to 1\}, \{a \to 1, b \to 1, c \to 1\})$	
$(\{a \to 1, b \to 1, c \to 0\}, \{a \to 1, b \to 1, c \to 0\})$	
$(\{a \to 1, b \to 1, c \to 0\}, \{a \to 1, b \to 1, c \to 1\})$	
$(\{a \to 1, b \to 0, c \to 1\}, \{a \to 1, b \to 0, c \to 1\})$	
$(\{a \to 1, b \to 0, c \to 1\}, \{a \to 1, b \to 1, c \to 1\})$	
$(\{a \to 0, b \to 1, c \to 1\}, \{a \to 0, b \to 1, c \to 1\})$	
$(\{a \to 0, b \to 1, c \to 1\}, \{a \to 1, b \to 1, c \to 1\})$	
$(\{a \to 1, b \to 1, c \to 1\}, \{a \to 1, b \to 1, c \to 1\})$	

$\mathcal{U}$

The subsequent computation for $\mathcal{T}$ produces the same results as the computation for $\mathcal{U}$ that we have just detailed, and is therefore omitted here. From the table above, we can observe that our evaluation process has successfully identified all pairs P, Q such that $P \subseteq Q$. At the same time we can see that $rel(non\_subset)$ will captures the fact that P can't be subset of Q if there exists an X such that $X \in P \land X \notin Q$

If we consider a predicate p which is true for $\{a, b, c\}$ and submit the query `subset (R,p)` then we will obtain every possible subset of the corresponding relation for the predicate p. The way this works is that the predicate constant p will match only with the rows from the subset relation that are supersets of $\{a, b, c\}$. Therefore, it will only match with the rows that have $\{a \to 1, b \to 1, c \to 1\}$. This will result in the following relation:

R	
$\{a \to 0, b \to 0, c \to 0\}$	
$\{a \to 1, b \to 0, c \to 0\}$	
$\{a \to 0, b \to 1, c \to 0\}$	
$\{a \to 0, b \to 0, c \to 1\}$	After evaluating `subset(R,p)`
$\{a \to 1, b \to 1, c \to 0\}$	
$\{a \to 1, b \to 0, c \to 1\}$	
$\{a \to 0, b \to 1, c \to 1\}$	
$\{a \to 1, b \to 1, c \to 1\}$	

As we can clearly see here, R represents the sets $\emptyset$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, and $\{a, b, c\}$.

Now, let us consider two other predicates, p and q, with the following relations:

$$rel(p) = \{a\}, \quad rel(p') = \{a\}, \quad rel(q) = \emptyset, \quad rel(q') = \{a\}.$$

This means that p is true for a, while q is undefined for a. We aim to verify whether `subset(p,q)` holds.

If we attempt to match against any tuple in $rel(subset)$, the match will fail. Although p can match any set S such that $S \supseteq \{a \to 1\}$, $rel(q) = \emptyset$. From the table above, it is clear that no pair P, Q exists such that $P \neq \emptyset$ and $P \subseteq Q$. Therefore, `subset(p,q)` does not hold.

On the other hand, if we try to match against the same tuples in $rel(subset')$, we encounter relaxed constraints that allow Q to match successfully against any relation where a is at least undefined. As we can see, q satisfies this condition exactly. The fact that we definitely know p is true for a, combined with the fact that q is undefined for a, results in assigning the undefined value to `subset(p,q)`.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we introduced a bottom-up evaluation method for second-order Datalog programs with negation under well-founded semantics. Our approach provides a structured way to handle negation and higher-order logic, addressing key challenges in extending evaluation techniques beyond first-order logic. We also developed a Python implementation of this method, which is available for reference and further exploration at `https://github.com/antonis96/sodn-evaluator`.

While the current implementation meets its main goals, it faces challenges when evaluating programs that produce large relations, particularly those involving second-order constructs. Operations such as joins become computationally expensive as these relations grow, requiring significant time and memory to process extensive sets of mappings.

Example 5.1. *Consider the following program $\mathcal{P}$ with $\mathcal{H}_{\mathcal{U}} = \{a_1, \ldots, a_k\}$:*

```
p(R) ← R(X₁), R(X₂), ..., R(Xₙ).
q(R) ← not p(R).
```

where $k \geq n$. In this case, the relation $rel(p)$ consists of all possible non-empty sets of mappings where each element of $\mathcal{H}_{\mathcal{U}}$ is assigned the value 1:

R
$\{(a_1) \to 1\}$
$\{(a_2) \to 1\}$
$\vdots$
$\{(a_k) \to 1\}$
$\{(a_1) \to 1, (a_2) \to 1\}$
$\vdots$
$\{(a_1) \to 1, (a_k) \to 1\}$
$\vdots$
$\{(a_1) \to 1, (a_2) \to 1, \ldots, (a_k) \to 1\}$

The number of elements in $rel(p)$ is given by:

$$|rel(p)| = 2^k - 1$$

since it contains every non-empty subset of mappings from $\mathcal{H}_{\mathcal{U}}$ where each element is mapped to 1.

Now, consider the relation $rel(q)$, which represents the complement of $rel(p)$. This means that $rel(q)$ must contain all sets of mappings where at least one value in the mapping is flipped (i.e., changed from 1 to 0). For every subset in $rel(p)$, multiple alternative versions appear in $rel(q)$ — one for each way of flipping at least one value.

Consequently, the size of $rel(q)$ is given by:

$$|rel(q)| = \sum_{i=1}^{k} \binom{k}{i}(2^i - 1) = 2^k - 1 + \sum_{i=1}^{k-1} \binom{k}{i}(2^i - 1) \gg |rel(p)|$$

This formula accounts for every subset of $\mathcal{H}_{\mathcal{U}}$ where at least one element has been flipped. The rapid growth of $rel(q)$, significantly outpacing that of $rel(p)$, illustrates how second-order constructs can lead to an explosion in the number of generated facts. This exponential increase in the number of mappings makes evaluation computationally expensive, particularly as k increases.

One observation is that by restricting a program to safe rules—where every variable in the head appears in at least one positive literal in the body—we can reduce computation time. Safe rules ensure that intermediate relations remain small, thereby avoiding the generation of unnecessarily large complements.

Example 5.2. *Consider a program $\mathcal{P}$ that includes the rule:*

```
k(R) ← p(R), not q(R).
```

Suppose the following relations are derived during evaluation:

$rel(p)$
$\{(a) \to 1\}$
$\{(b) \to 1\}$
$\{(c) \to 1\}$
$\{(a) \to 1, (b) \to 1\}$
$\{(a) \to 1, (c) \to 1\}$
$\{(b) \to 1, (c) \to 1\}$
$\{(a) \to 1, (b) \to 1, (c) \to 1\}$

$rel(q)$
$\{(a) \to 1, (b) \to 1\}$
$\{(a) \to 1, (c) \to 1\}$
$\{(b) \to 1, (c) \to 1\}$
$\{(a) \to 1, (b) \to 1, (c) \to 1\}$

Rather than computing the full complement of $rel(q)$ which can produce an excessively large intermediate relation—we directly derive $rel(k)$ by taking the difference:

$$rel(k) = rel(p) \setminus rel(q).$$

This produces:

$rel(k)$
$\{(a) \to 1\}$
$\{(b) \to 1\}$
$\{(c) \to 1\}$

Thus, $rel(k)$ contains only those tuples in $rel(p)$ that are not in $rel(q)$. This example demonstrates how safe rules help keep intermediate relation sizes manageable, thereby reducing computational overhead.

A further example highlights a limitation of bottom-up evaluation.

Example 5.3. *Consider a slightly modified version of the previous program:*

```
k(a).
p(R) ← R(X₁), R(X₂), ..., R(Xₙ).
q(a) ← not p(R), k(a).
```

Note that even though the variable `R` *in* `not p(R)` *is irrelevant to the final outcome (which depends solely on* `k(a)`*), bottom-up evaluation still processes* `not p(R)` *over all possible values. This behavior highlights a known inefficiency in bottom-up approaches.*

Future work could also focus on integrating semi-naive evaluation and the Magic Sets transformation, two established techniques for optimizing logic program evaluation. Semi-naive evaluation reduces runtime by avoiding redundant computations in iterative rule processing, considering only newly derived facts at each step. Incorporating this technique could significantly improve efficiency, particularly for programs generating large relations.

The Magic Sets transformation [8] offers another complementary optimization by rewriting rules to focus computations on query-relevant portions of the program. This approach minimizes redundant work and irrelevant computations, making it especially effective for programs with extensive relations. When combined with semi-naive evaluation, Magic Sets could further enhance performance by dynamically refining the scope of evaluation and preventing unnecessary recomputations. Together, these techniques could make the framework more efficient and scalable.

In summary, this thesis provides a strong foundation for evaluating second-order Datalog programs with negation. While challenges remain, particularly in handling large and complex programs, the framework offers many useful features. With future advancements like query-focused optimizations, it has the potential to become a robust and efficient tool for both research and practical applications.

BIBLIOGRAPHY

[1] Angelos Charalambidis, Konstantinos Handjopoulos, Panos Rondogiannis, and William W. Wadge. Extensional Higher-Order Logic Programming. *CoRR*, abs/1106.3457, 2011. URL: http://arxiv.org/abs/1106.3457.

[2] Angelos Charalambidis and Panos Rondogiannis. Constructive Negation in Extensional Higher-Order Logic Programming. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 12–21. AAAI Press, 2014.

[3] Panos Rondogiannis and Ioanna Symeonidou. Extensional Semantics for Higher-Order Logic Programs with Negation. *CoRR*, abs/1701.08622, 2017. URL: http://arxiv.org/abs/1701.08622.

[4] Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Approximation Fixpoint Theory and the Well-Founded Semantics of Higher-Order Logic Programs. *CoRR*, abs/1804.08335, 2018. URL: http://arxiv.org/abs/1804.08335.

[5] David Kemp, Peter Stuckey, and Divesh Srivastava. Magic Sets and Bottom-up Evaluation of Well-Founded Models. In *Proceedings of the 1991 International Symposium on Logic Programming*, 1991.

[6] Allen Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185-221, 1993. URL: https://doi.org/10.1016/0022-0000(93)90024-Q.

[7] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970. URL: https://doi.org/10.1145/362384.362685.

[8] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[9] D. Carral, M. Keeler, and B. C. Grau. Chasing existential rules with sets. Proceedings of the AAAI Conference on Artificial Intelligence, 2019. Available at: https://www.ijcai.org/proceedings/2019/0225.pdf.

[10] S. Gaggl, T. Linsbichler, and S. Woltran. ASP with sets: Efficient grounding and reasoning. Theory and Practice of Logic Programming, 2022. Available at: `https://www.ijcai.org/proceedings/2022/0365.pdf`.

[11] A. Luppnow. Bottom-Up Evaluation of HiLog in the Context of Deductive Database Systems. Master's Thesis, University of Cape Town, 1998. URL: `https://open.uct.ac.za/server/api/core/bitstreams/80408735-3f7a-4524-9cad-ad51fcf6ba1c/content`