The Expressive Power of Higher-Order Datalog with Negation

Charalampos Kostopoulos
AL1.19.0014

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

It is a well-known result that Datalog, with the added assumption that the input databases are ordered, can express exactly those queries that belong in class PTIME. Recently, this result was extended for higher-order Datalog that does not contain negation. It appears that an increase in the order allowed in the programs results in a direct increase in expressiveness. This thesis studies the expressiveness of higher-order Datalog when we introduce negation. More specifically, we present a study of the expressiveness of different higher-order Datalog language fragments where each time a subset of features is allowed from negation, partially-applied expressions as predicate arguments, and higher-order existential variables. For each such choice of features, we investigate the resulting expressiveness as we increase the order for the types we allow in our programs.

It follows that the original expressiveness result is not changed when we just introduce negation. However, in the case that we restrict partial application from our syntax, there is a big difference. Partially-applied expressions can alone give the maximum expressiveness achieved by higher-order Datalog while in their absence only negation paired with higher-order existential variables can achieve the same result. Finally, all other choices of features result in a collapse in PTIME for the problems the language fragments can express regardless of the order we allow in our programs.

ΣΥΝΟΨΗ

Είναι γνωστό αποτέλεσμα στη βιβλιογραφία ότι η Datalog με την υπόθεση ότι οι βάσεις δεδομένων εισόδου είναι διατεταγμένες, μπορεί να εκφράσει όλα εκείνα τα ερωτήματα που ανήκουν στην κλάση πολυπλοκότητας PTIME. Πρόσφατα, αυτό το αποτέλεσμα επεκτάθηκε και σε προγράμματα Datalog υψηλής τάξης που δεν περιέχουν άρνηση και αποδείχτηκε οτι η εκφραστικότητα αυξάνεται αυστηρά καθώς αυξάνεται και η τάξη των προγραμμάτων. Αυτή η διπλωματική μελετάει την εκφραστικότητα προγραμμάτων υψηλής τάξης Datalog όταν επιτρέπουμε άρνηση. Πιο συγκεκριμένα παρουσιάζεται μια ανάλυση της εκφραστικότητας διαφόρων εκδόσεων της υψηλής τάξης Datalog όπου κάθε φορά επιτρέπουμε ένα υποσύνολο από χαρακτηριστικά όπως η άρνηση, μερικώς εφαρμοσμένες εκφράσεις ως ορίσματα και υπαρξιακές μεταβλητές υψηλής τάξης. Σε κάθε επιλογή τέτοιων χαρακτηριστικών μελετάμε την επίδραση της αύξησης της επιτρεπόμενης τάξης των προγραμμάτων στην εκφραστικότητα της γλώσσας.

Προκύπτει ότι η εκφραστικότητα δεν αυξάνεται περαιτέρω όταν εισάγουμε τον τελεστή της άρνησης. Αν όμως δεν επιτρέψουμε μερικώς εφαρμοσμένες εκφράσεις στα προγράμματά μας τότε υπάρχει σημαντική διαφορά. Οι μερικώς εφαρμοσμένες εκφράσεις στα προγράμματα δίνουν από μόνες τους την μέγιστη εκφραστικότητα της γλώσσας, ενώ αν απουσιάζουν χρειαζόμαστε άρνηση με υπαρξιακές μεταβλητές για να πετύχουμε την ίδια εκφραστικότητα. Σε όλες τις άλλες περιπτώσεις η εκφραστικότητα της γλώσσας πέφτει στην κλάση PTIME ανεξαρτήτως της τάξης που επιτρέπουμε να έχουν οι τύποι στα προγράμματά μας.

CONTENTS

CHAPTER 1 INTRODUCTION

In the functional programming paradigm the use of first-class functions that are passed as arguments and returned as results have unlocked significant advantages in the versatility and compactness of writing code. Furthermore, it has been demonstrated in ([7]) that when we restrict the language to only accept read-only programs, where creating new data structures is not possible, the use of higher-order functions increases the complexity class of the problems that can be expressed.

In the logic programming paradigm, while these languages are generally expected to be first-order, there has been extensive work to define and develop general-purpose higher-order languages. In such cases, even the first-order variant is Turing-Complete. However, this is not the case with Datalog, which does not allow the use of function symbols in its syntax [5]. Datalog is a declarative logic programming language that syntactically serves as a subset of Prolog. Datalog employs a bottom-up evaluation model, as opposed to the top-down evaluation model of Prolog, resulting in significantly different behavior and properties. It is often employed as a query language for deductive databases.

Example 1.1. An example of a Datalog program using a slightly different syntax than standard Datalog.

```
edge a b.
edge b c.
path X Y   ←   (edge X Y).
path X Y   ←   (edge X Z),(path Z Y).
```

It is shown that Datalog captures only PTIME ([8]) or in other words it can express all such queries that belong to this complexity class. However, this capability is achieved under the assumption that the input database provided to the query program is ordered. An ordered database is one that includes a predicate implying a total ordering relation among its elements. Without this assumption, Datalog is incapable of deciding problems even within PTIME such as deciding whether the total number of elements of the database is odd or even.

Example 1.2. A predicate like `next` needs to be present in the input database.

```
next a b.
next b c.
next c d
...
```

In this thesis we study the higher-order version of Datalog in the form that is introduced in [1]. The syntax of this language is similar to classic Datalog but the formal parameters of a program rule can also be distinct predicate variables. Furthermore, we can use partially applied predicates as new predicates.

Example 1.3. A program of higher-order Datalog.

```
edge a b.
edge b c.
closure R X Y       ←  (R X Y).
closure R X Y       ←  (R X Z),(closure R Z Y).
path X Y            ←  (closure edge X Y ).
reverse R X Y       ←  (R Y X).
reversed_path X Y   ←  (closure (reverse edge) X Y).
```

The classical result of the expressiveness of Datalog is extended for the higher-order setting in [1]. In an analogous sense to the work done in [7] for the read-only functional languages, an increase in the program order gives an increase in expressiveness allowing for a broader class of problems to be expressed. For example, second-order Datalog is shown to capture exactly EXPTIME, third-order Datalog captures $2-$EXPTIME and so on. However a version of *positive* Datalog is considered in that work. Specifically the syntax of the language does not allow a negation operator. It also does not allow higher-order existential variables. These are variables that appear only in the body of a program rule and not in the head.

Example 1.4. A program of higher-order Datalog that contains existential higher-order variables. Variable `M` is existential and is used to retrieve the "previous" first-order relation.

```
last X.
recursive_p N  ←  (predecessor N M), (recursive_p M).
recursive_p N  ←  (edge_condition N).
test           ←  (recursive_p last).
```

This raises questions about what happens when we introduce additional features, such as existential higher-order variables and negation. Do these features enhance expressiveness? Furthermore, a second question emerges regarding the significance of having partial application in a higher-order language. Could a predicate like `'predecessor'`, as in the previous example, be defined and function in the intuitive manner we expect, effectively simulating a counting of an exponential number of elements in positive Datalog? Such predicates have already been defined in [1] but they rely on partial application and not existential variables.

Example 1.5. Simulating counting by using partially applied predicates.

```
last X.
recursive_p N       ←  (recursive_p (predecessor N)).
recursive_p N       ←  (edge_condition N).
test                ←  (recursive_p last).

predecessor N X V   ←  ...
```

partial application	negation	h.o. existential variables	order 1	order 2	order 3	$\cdots$	order ∞
YES	X	X	PTIME	EXPTIME	$2-$EXPTIME	$\cdots$	ELEM.
NO	YES	YES	PTIME	EXPTIME	$2-$EXPTIME	$\cdots$	ELEM.
NO	YES	NO	PTIME	PTIME	PTIME	$\cdots$	PTIME
NO	NO	X	PTIME	PTIME	PTIME	$\cdots$	PTIME

It turns out that partial application is not only necessary but also sufficiently powerful to maintain the established level of expressiveness. The addition of the other two features does not lead to any further increase in expressiveness. Furthermore, if we wish to restrict partial application, both higher-order existential variables and negation must be permitted to preserve this level of expressiveness. Any other alternatives result in a return to PTIME [1], regardless of the order of the programs, which aligns precisely with classic first-order positive Datalog.

The results of this thesis are summarized in a concise table here. In the table, "YES" and "NO" indicate the inclusion and exclusion, respectively, of the mentioned feature, while "X" signifies that inclusion or exclusion is irrelevant. We use "X" to condense multiple rows into one for brevity. Since we introduce negation, it necessitates the adoption of a different, more general semantics for our language known as Well-Founded semantics. This semantics is based on a three-valued truth model (*true*, *false*, *undef*). The extension of such a semantics model to a higher-order setting is a recent development presented in [2].

The thesis is organized as follows.

- Chapter 2 is dedicated to establishing the syntax of higher-order Datalog, following the steps outlined in [1]. In this chapter, we introduce the negation operator and remove the restriction of using only formal variables in the body of program rules. This syntax serves as the foundation, and we subsequently restrict it to create various fragments. Additionally, we define the concept of order for the types in the language.

- In Chapter 3, we provide the semantics for our language, which is based on the extension of Well-Founded semantics for a higher-order language. We draw upon the work in [2], presenting or extending relevant definitions and theorems as needed. Furthermore, we define what we mean when we state that Datalog decides a problem, specifically whether a string belongs to a formal language L.

- Chapter 4 presents the expressiveness of all higher-order Datalog variants with partial application in a concise manner, as they are all equivalent, as shown in the first row of Table 1.

- Chapter 5 is dedicated to proving the result corresponding to row 2 of the table, which is the only combination without partial application that restores the hierarchy. To demonstrate this result, we need to establish two necessary statements:

    – Firstly, we aim to show that any language L that can be decided by a $(k-1)$-exponential-time Turing machine can also be decided by a k-order Datalog program belonging to the specified language fragment. To achieve this, we

[1]The third row is not shown in this work.

  present a simulation of the Turing Machine. The key challenge lies in efficiently encoding exponentially large numbers using higher-order predicates while abiding to the rules of the restricted syntax.

  – Secondly, we demonstrate that for every k-order Datalog program from the designated fragment set, the computation of its minimal Well-Founded model can be accomplished by a $(k-1)$-exponential-time bounded Turing machine. We provide a high-level description of a multi-tape Turing Machine capable of calculating the model within the required time bounds. In fact, we had to provide such a machine in Chapter 4 for the most general version of the higher-order Datalog we study.

  These two facts together imply that k-order Datalog can decide any language that falls within $(k-1) - \mathsf{EXPTIME}$ and no language that falls outside.

- Chapter 6 is dedicated to demonstrating that the fragments mentioned in row 4 of the table precisely capture PTIME and that no further gains can be achieved by increasing the program order. This endeavor will involve a multi-step approach, with a crucial component being the influential result presented in [3]. This result enables us to adopt a more syntactical approach to our semantics, particularly for programs that do not incorporate the negation operator.

- Chapter 7 is dedicated to a discussion of future work in this field. A significant portion of this discussion is about the fragments in the third row of the table, specifically the ones that allows negation without existential higher-order variables. While it is expected that these fragments, for any program order, should fall within PTIME, a more intricate approach was required to formally establish this. In this chapter, we briefly describe an approach one can take to demonstrate this, along with the challenges that arise from it. While we do not include detailed proof in this work, we leave it for a future article.

CHAPTER 2
SYNTAX AND ORDER

The language we will use is a function-free subset language used in [2]. In this chapter, we will present the syntax for the language in its most general version (e.g negation, existential higher-order variables, partial application).

2.1 The Syntax

The syntax follows from the classical syntax of Datalog with an important restriction that every higher-order iteration of Datalog must abide by.
Every argument of predicate type in the head of a rule must be a variable. Furthermore, all such variables must be distinct.
Without this restriction, monotonicity in positive higher-order Datalog is not preserved. More importantly removing the restriction can lead to non-extensional models for a given program in the higher-order setting.

Example 2.1. Consider the following example program that does not follow this rule.

```
r a.
q a.
p q.
p R   ←   R b
```

In the minimum model, it is $\mathsf{q} = \mathsf{r} = \{\mathsf{a}\}$ which makes both predicates equivalent or in other words they have the same extensional meaning as they represent the same relation. The problem arises from the fact that we also get $(\mathsf{p}\ \mathsf{q}) =$ *true* and $(\mathsf{p}\ \mathsf{r}) =$ *false*. In an extensional model the predicate p should behave the same when given inputs that are equivalently extensionaly.

Following in the steps of [1] we present the simple type system of the language first. There are only two base types: o which corresponds to the boolean domain and ι which corresponds to the domain of individuals. The composite types are divided into two categories. The predicate types π and the argument types ρ. Only an argument-type variable can appear as an argument of a predicate in the body of a program rule.

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

$$\pi := o \mid (\rho \to \pi)$$
$$\rho := \iota \mid (\rho \to \pi)$$

Notice that we don't allow type o to appear as an argument type. This is to comply with the classical results for first-order Datalog where predicates are not allowed to have as arguments, expressions that are other fully applied predicates. It can be shown that the inclusion of o to the argument type does not add to the expressive capabilities in any of the cases we consider.

Next, we define the alphabet of the language. We will use a subset of the language described in [2] since we will need to also define a negation operator and later the Well-Founded semantics for our language.

Definition 2.2. The *alphabet* consists of:

1. *Predicate variables* of every predicate type π (denoted by capital letters such as P and Q).
2. *Predicate constants* of every predicate type π (denoted by lowercase letters such as p and q).
3. *Individual variables* of type ι (denoted by capital letters such as X and Y).
4. *Individual constants* of type ι (denoted by lowercase letters such as a and b).
5. The following *logical constant symbols*: the equality constant $\approx$ for comparing expressions of type ι; the conjuction symbol $\wedge$; the inverse implication constant $\leftarrow$; the negation constant operator $\sim$.
6. The parentheses "(" and ")".

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 is a term of type $\rho \to \pi$ and E_2 a term of type ρ then $(\mathsf{E}_1\ \mathsf{E}_2)$ is a term of type π. In the case of applying multiple arguments, we will use the left associativity rule to omit parenthesis when possible.

Next, we define the set of expressions for our language.

Definition 2.4. The set of *expressions* of Higher-Order Datalog is defined as follows:

- A term of type ρ is an expression of type ρ;
- If E_1 and E_2 are terms of type ι, then $(\mathsf{E}_1 \approx \mathsf{E}_2)$ is an expression of type o.
- If E is an expression of type o then $\sim \mathsf{E}$ is an expression of type o.

Expressions (respectively terms) that have no variables will often be referred to as *ground expressions* (respectively *ground terms*). Expressions of type o that are either terms or have the form $\mathsf{E}_1 \approx \mathsf{E}_2$ where $\mathsf{E}_1, \mathsf{E}_2$ terms of type ι, will often be referred to as *atoms*. Additionally, we will call *literals* all expressions that are atoms or negated atoms. To denote that an expression E has type ρ we will often write $\mathsf{E} : \rho$.

Definition 2.5. A *rule* of Higher-Order Datalog is a formula $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$, where p is a predicate constant of type $\rho_1 \to \cdots \to \rho_n \to o$, $\mathsf{V}_1, \ldots, \mathsf{V}_n$, $n \geq 0$, are *distinct* [1] argument variables of types $\rho_1, \ldots, \rho_n$ respectively, and for every $i \in \{1, \cdots, m\}$, L_i is a literal. More specifically

$$\mathsf{L}_i = \begin{cases} \mathsf{E}_i \\ \sim\!\mathsf{E}_i \end{cases}$$

where E_i is an atom.

The term $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n$ is called the *head* of the rule, the variables $\mathsf{V}_1, \ldots, \mathsf{V}_n$ are the *formal parameters* of the rule and the conjunction $\mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$ is its *body*.

Notice how the definitions of the terms and atoms do not include the negation constant and literals can have at most one negation at the top level of their expression. Therefore, the rule definition restricts nested negations to appear or negation at a deeper level of an expression. Finally, a program rule will be called a *positive* rule if the negation constant does not appear in its body. A *positive* higher-order Datalog program is one where every one of its rules is positive.

It is easy to see that this syntax is a subset of the higher order language "HOL" defined in [2] which is the work we will also use to define the semantics.

We will borrow some features of Prolog's syntax while writing examples and when presenting the code for the simulations in subsequent chapters for readability and familiarity reasons. Specifically, instead of the conjunction symbol, we will use commas to separate each expression in the body and end it with a full stop.

Example 2.2. For example, we will write

$$\texttt{pred}_{k+1}\ \texttt{N M} \quad \leftarrow \quad (\texttt{non\_zero}_{k+1}\ \texttt{N}),(\texttt{hpred}_1\ \texttt{N M last}_k).$$

Now, we define the order that a type has.

Definition 2.6. The *order* of a type is recursively defined as follows:

$$\begin{aligned} \mathit{order}(\iota) &= 0 \\ \mathit{order}(o) &= 0 \\ \mathit{order}(\rho_1 \to \cdots \to \rho_n \to o) &= 1 + \mathit{max}(\{\mathit{order}(\rho_i) \mid 1 \leq i \leq n\}) \end{aligned}$$

The order of a predicate constant (or variable) is the order of its type.

Then we will define fragments of the general language based on the order as follows.

Definition 2.7. For all $k \geq 1$, *k-order Datalog* is the fragment of Higher-Order Datalog in which all predicate constants have order less than or equal to k and all predicate variables have order less than or equal to $k - 1$.

This is one of the directions we restrict each fragment. The other kind of restriction is based on restricting the syntax presented in this section. The next section establishes that.

[1] All the formal parameters are distinct variables (i.e., for all i, j such that $1 \leq i, j \leq n$ and $i \neq j$, $\mathsf{V}_i \neq \mathsf{V}_j$).

2.2 Higher-Order Datalog fragments

There are three syntactical restrictions considered in this work. We choose between allowing our programs to have:

- The negation $\sim$ constant e.g.

```
p Q  ←  ∼ (Q a).
```

- Partially applied predicates in an argument position of a term e.g.

```
r X    ←  (q (p r) z).
p R Z  ←  (R a), (Z b).
```

  Partial application allows our terms to have syntactic trees of arbitrary height. Removing this option forces every term to have a syntactic tree of restricted height.

- Higher order existential variables in the body of rules. These are variables of at least type order 1 that appear only in the body of a rule and not also in the head e.g.

```
p X  ←  (R X).
```

For each combination of choice for these restrictions, we create a fragment of higher-order Datalog. By also choosing the program order k for $k \geq 1$ on top of the previous restrictions we get a language fragment of k-order Datalog.

In the case of restricting partial application, we formally consider a modified definition of the notions of atoms and literals. More formally we have:

Definition 2.8. A *simple atom* is an expression of type o and of either the form $(\mathsf{E}_1 \approx \mathsf{E}_2)$ or of the form

$$(\mathsf{E}_0\ \mathsf{E}_1 \ldots \mathsf{E}_{n-1})$$

where each E_i is either a variable or a constant.

Definition 2.9. A *simple literal* is either a simple atom or the negation of a simple atom.

Definition 2.10. A *simple rule* of Higher-Order Datalog is a formula $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$, where p is a predicate constant of type $\rho_1 \to \cdots \to \rho_n \to o, \mathsf{V}_1, \ldots, \mathsf{V}_n$, $n \geq 0$, are argument variables of types $\rho_1, \ldots, \rho_n$ respectively, and for every $i \in \{1, \cdots, m\}$, L_i is a simple literal.

Definition 2.11. A *simple* program or otherwise a program devoid of partial application is one that consists of only simple rules.

Definition 2.12. Let C a subset of the symbols $\{\lambda, \neg, \exists\}$. Then $\mathcal{H}_k^C$ is the fragment of k-order Datalog that syntactically restricts what does not appear in C. Specifically, in the superscript

- If λ does not appear then only simple programs are allowed.
- If $\neg$ does not appear then the negation constant is not allowed.
- If $\exists$ does not appear then higher-order existential variables are not allowed.

For example $\mathcal{H}_k^{\lambda,\neg}$ is the fragment of k-order Datalog where existential variables are not allowed but negation and partial application are.

CHAPTER 3 SEMANTICS AND DECISION PROBLEMS

In this chapter, we will provide the semantics of our source language. The semantics will be drawn from [2] which describes a more general higher-order language that has a richer syntax and also allows function symbols therefore it is Turing complete. This semantics is called Well-Founded semantics (WFS) and is a generalization of the classical WFS model to the higher-order setting.

Because we will also consider language fragments that don't contain the negation operator we will present a semantics for positive higher-order Datalog as well, like shown in [1]. Changing semantics for higher-order Datalog is not a part of investigating language expressiveness. In fact, we only consider different fragments based on syntactical restrictions and the type order. The reason for also introducing positive higher-order semantics is to simplify the process of handling higher-order Datalog fragments that restrict the negation operator. As we will show later the WFS model of a positive program and the minimum model under classical positive semantics coincide in a more general sense even though they are defined in different truth domains. It follows as well that all ground terms of type o in *positive* under both semantics get assigned the same truth values (*false*, *true*) despite the fact that Well-Founded semantics introduces a new third truth value called *undef*. We will exploit this fact to simplify the proofs for those fragments and use the pre-existing work done for positive higher-order Datalog.

Finally, we will define the notion of deciding a language through Datalog and establish the background that will be used to distinguish the expressive capabilities of each Datalog fragment we investigate.

3.1 Positive Higher-Order Datalog Semantics

The semantics of positive Higher-Order Datalog, which is based on the ideas initially proposed in [10] and [4] is an extensional model of semantics where we treat program predicates as monotone relations or otherwise monotone functions to the set $\{false, true\}$.

For the types, we define recursively the semantics $[\![\rho]\!]$ of a type ρ and we also define at the same time a corresponding partial order $\leq_\rho$ on the elements of $[\![\rho]\!]$. We adopt the usual ordering of the truth values *false* and *true*, i.e. $false < true$. Let A and B be partial orders, we write $[A \overset{m}{\to} B]$ to denote the set of all monotone functions from A to B. For a function f in this set it must hold that $\forall a, b \in A$ if $a \leq b$ then $f(a) \leq f(b)$.

Definition 3.1. Let P be a program. The *Herbrand universe* U_{P} of P is the set of all constant elements that appear in it. Then:

- $[\![\iota]\!] = U_{\mathsf{P}}$ and $\leq_\iota$ is the trivial partial order that relates every element of U_{P} to itself
- $[\![o]\!] = \{\mathit{false}, \mathit{true}\}$ and $\leq_o$ is the partial order $\leq$ on truth values
- $[\![\rho \to \pi]\!] = [[\![\rho]\!] \overset{m}{\to} [\![\pi]\!]]$ and $\leq_{\rho\to\pi}$ is the partial order defined as follows: for all $f, g \in [\![\rho \to \pi]\!]$, $f \leq_{\rho\to\pi} g$ iff $f(d) \leq_\pi g(d)$ for all $d \in [\![\rho]\!]$.

We proceed to define Herbrand interpretations and states.

Definition 3.2. A *Herbrand interpretation I of a program* P is a function that assigns:

- to each individual constant c that appears in P, the element $I(\mathsf{c}) = \mathsf{c}$;
- to each predicate constant $\mathsf{p} : \pi$ that appears in P, an element $I(\mathsf{p}) \in [\![\pi]\!]$;

Definition 3.3. A *Herbrand state s* of a program P is a function that assigns to each argument variable V of type ρ, an element $s(\mathsf{V}) \in [\![\rho]\!]$.

In the following, $s[\mathsf{V}_1/d_1, \ldots, \mathsf{V}_n/d_n]$ is used to denote a state that is identical to s the only difference being that the new state assigns to each V_i the corresponding value d_i.

Definition 3.4. Let P be a program, I a Herbrand interpretation, and s a Herbrand state of P. Then, the semantics of expressions is defined as follows:

- $[\![\mathsf{V}]\!]_s(I) = s(\mathsf{V})$;
- $[\![\mathsf{c}]\!]_s(I) = I(\mathsf{c})$;
- $[\![\mathsf{p}]\!]_s(I) = I(\mathsf{p})$;
- $[\![(\mathsf{E}_1\ \mathsf{E}_2)]\!]_s(I) = [\![\mathsf{E}_1]\!]_s(I)([\![\mathsf{E}_2]\!]_s(I))$;
- $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_s(I) = true$ if $[\![\mathsf{E}_1]\!]_s(I) = [\![\mathsf{E}_2]\!]_s(I)$ and *false* otherwise.

For ground expressions E we will often write $[\![\mathsf{E}]\!](I)$ instead of $[\![\mathsf{E}]\!]_s(I)$ since in this case the meaning of E is independent of s. The *model* of the program is defined as follows:

Definition 3.5. Let P be a program and M a Herbrand interpretation of P. Then, M is a *Herbrand model* of P iff for every rule $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{E}_1 \wedge \cdots \wedge \mathsf{E}_m$ in P and for every Herbrand state s, if for all $i \in \{1, \ldots, m\}$, $[\![\mathsf{E}_i]\!]_s(M) = \mathit{true}$ then $[\![\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n]\!]_s(M) = \mathit{true}$ or equivalently:

$$\bigwedge\{[\![\mathsf{E}_1]\!]_s(M), \cdots, [\![\mathsf{E}_m]\!]_s(M)\} \leq_o [\![\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n]\!]_s(M)$$

where $\bigwedge$ is the maximum lower bound operator and in this case, it operates on type o.

We denote the set of Herbrand interpretations of a program P with $\mathcal{I}_{\mathsf{P}}$, and define a partial order on $\mathcal{I}_{\mathsf{P}}$ as follows: for all $I, J \in \mathcal{I}_{\mathsf{P}}$, $I \leq_{\mathcal{I}_{\mathsf{P}}} J$ iff for every predicate constant $\mathsf{p} : \pi$ that appears in P, $I(\mathsf{p}) \leq_{\pi} J(\mathsf{p})$. We denote by $\bigvee$ the least upper bound operation and by $\bot_{\mathcal{I}_{\mathsf{P}}}$ the least element of the lattice, with respect to $\leq \mathcal{I}_{\mathsf{P}}$. Intuitively, $\bot_{\mathcal{I}_{\mathsf{P}}}$ assigns to every program predicate in P the empty relation or the function which always points to *false* for input.

We will need to define the *immediate consequence operator* for Higher-Order Datalog programs, which generalizes the corresponding operator for classical Datalog.

Definition 3.6. Let P be a program. The mapping $T_{\mathsf{P}} : \mathcal{I}_{\mathsf{P}} \to \mathcal{I}_{\mathsf{P}}$ is called the *immediate consequence operator for* P and is defined for every predicate constant $\mathsf{p} : \rho_1 \to \cdots \to \rho_n \to o$ and $d_i \in [\![\rho_i]\!]$ as:

$$T_{\mathsf{P}}(I)(\mathsf{p})\, d_1 \cdots d_n = \begin{cases} \textit{true}, & \text{if there exists a clause } \mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{E}_1 \wedge \cdots \wedge \mathsf{E}_m \text{ in } \mathsf{P} \text{ and} \\ & \text{a Herbrand state } s\text{, such that } [\![\mathsf{E}_i]\!]_{s[\mathsf{V}_1/d_1,\ldots,\mathsf{V}_n/d_n]}(I) = \textit{true} \\ & \text{for all } i \in \{1, \ldots, m\} \\ \textit{false}, & \text{otherwise.} \end{cases}$$

Define now the following sequence of interpretations:

$$\begin{array}{lcl} T_{\mathsf{P}} \uparrow 0 & = & \bot_{\mathcal{I}_{\mathsf{P}}} \\ T_{\mathsf{P}} \uparrow (n+1) & = & T_{\mathsf{P}}(T_{\mathsf{P}} \uparrow n) \\ T_{\mathsf{P}} \uparrow \omega & = & \bigvee \{T_{\mathsf{P}} \uparrow n \mid n < \omega\} \end{array}$$

We then have the following theorem:

Theorem 3.1. *Let P be a program and let $M_{\mathsf{P}} = T_{\mathsf{P}} \uparrow \omega$. Then, M_{P} is the least Herbrand model of P and the least fixpoint of T_{P} (with respect to the ordering relation $\leq_{\mathcal{I}_{\mathsf{P}}}$).*

This definition concludes the presentation of the semantics for positive Higher-Order Datalog. Next, we will proceed with the semantics with negation under the Well-Founded model of semantics.

3.2 Well-Founded Semantics for Negative Higher-Order Datalog

We present now the semantics for the version of higher-order we consider in this work and which contains a negation constant. The first and only work to our knowledge that extends the Well-Founded model of logic programming to a higher-order language setting is [2]. This is the work used here to define the semantics for our (subset) language and is strongly recommended that the reader studies that work for a deeper understanding of the following definitions. In the context of this thesis, we require a **constructive way** to produce the minimal now model, like the immediate consequence operator T_{P} in positive Datalog, in order to argue about the complexity of an algorithm that finds it.

3.2.1 The three-valued truth model

The boolean domain in contrast to the positive case is now three-valued. Predicates can't have the meaning of monotone functions and the previously defined immediate consequence operator does not always lead to a fix-point. In this new setting, predicates get the meaning of **Fitting-monotone** functions and a new partial order is introduced alongside the classic one, by the relation $\preceq$ which represents *information* or *Fitting* ordering as it is known.

Definition 3.7. Let D be a nonempty set. We define recursively for every type ρ the set of possible values-meaning that elements of this type can take by $[\![\rho]\!]_D$. The recursive definition is the following:

- $[\![o]\!]_D = \{\textit{false}, \textit{true}, \textit{undef}\}$. The partial order $\leq_o$ is the usual one induced by the ordering *false* $<_o$ *undef* $<_o$ *true*; the partial order $\preceq_o$ is the one induced by the ordering *undef* $\prec_o$ *false* and *undef* $\prec_o$ *true*.
- $[\![\iota]\!]_D = D$. The partial order $\leq_\iota$ is defined as $d \leq_\iota d$ for all $d \in D$. The partial order $\preceq_\iota$ is also defined as $d \preceq_\iota d$ for all $d \in D$.
- $[\![\iota \to \pi]\!]_D = D \to [\![\pi]\!]_D$. The partial order $\leq_{\iota\to\pi}$ is defined as follows: for all $f, g \in [\![\iota \to \pi]\!]_D$, $f \leq_{\iota\to\pi} g$ iff $f(d) \leq_\pi g(d)$ for all $d \in D$. The partial order $\preceq_{\iota\to\pi}$ is defined as follows: for all $f, g \in [\![\iota \to \pi]\!]_D$, $f \preceq_{\iota\to\pi} g$ iff $f(d) \preceq_\pi g(d)$ for all $d \in D$.
- $[\![\pi_1 \to \pi_2]\!]_D = [[\![\pi_1]\!]_D \xrightarrow{F.m} [\![\pi_2]\!]_D]$, namely the $\preceq$- monotonic functions from $[\![\pi_1]\!]_D$ to $[\![\pi_2]\!]_D$. These are the functions that it holds for all $d_1, d_2 \in [\![\pi_1]\!]_D$, $d_1 \preceq_{\pi_1} d_2$ then $f(d_1) \preceq_{\pi_2} f(d_2)$.

  The partial order $\leq_{\pi_1\to\pi_2}$ is defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]_D$, $f \leq_{\pi_1\to\pi_2} g$ iff $f(d) \leq_{\pi_2} g(d)$ for all $d \in [\![\pi_1]\!]_D$. The partial order $\preceq_{\pi_1\to\pi_2}$ is defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]_D$, $f \preceq_{\pi_1\to\pi_2} g$ iff $f(d) \preceq_{\pi_2} g(d)$ for all $d \in [\![\pi_1]\!]_D$.

Next, we have to define what an interpretation and a state are in this setting. The definitions are similar to the positive case.

Definition 3.8. An interpretation $\mathcal{I}$ consists of:

1. a nonempty finite set D called the *domain* of $\mathcal{I}$;
2. an assignment to each individual constant symbol c, of an element $\mathcal{I}(\mathsf{c}) \in D$;
3. an assignment to each predicate constant $\mathsf{p} : \pi$, of an element $\mathcal{I}(\mathsf{p}) \in [\![\pi]\!]_D$;

Definition 3.9. Let D be a nonempty set. A *state* s over D is a function that assigns to each argument variable V of type ρ to an element $s(\mathsf{V}) \in [\![\rho]\!]_D$.

We define: $\textit{true}^{-1} = \textit{false}$, $\textit{false}^{-1} = \textit{true}$ and $\textit{undef}^{-1} = \textit{undef}$. Finally, we define the semantics of the expressions of our language. We also define under which case an interpretation will be called a model of a program in the language.

Definition 3.10. Let D be a nonempty set, let $\mathcal{I}$ be an interpretation over D, and let s be a state over D. The semantics of expressions are then defined as:

1. $[\![\mathsf{c}]\!]_s(\mathcal{I}) = \mathcal{I}(\mathsf{c})$, for every individual constant c
2. $[\![\mathsf{p}]\!]_s(\mathcal{I}) = \mathcal{I}(\mathsf{p})$, for every predicate constant p
3. $[\![\mathsf{V}]\!]_s(\mathcal{I}) = s(\mathsf{V})$, for every argument variable R

4. $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(\mathcal{I}) = [\![\mathsf{E}_1]\!]_s(\mathcal{I})([\![\mathsf{E}_2]\!]_s(\mathcal{I}))$
5. $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_s(\mathcal{I}) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}_1]\!]_s(\mathcal{I}) = [\![\mathsf{E}_2]\!]_s(\mathcal{I}) \\ \mathit{false}, & \text{otherwise} \end{cases}$
6. $[\![(\sim\mathsf{E})]\!]_s(\mathcal{I}) = ([\![\mathsf{E}]\!]_s(\mathcal{I}))^{-1}$ in the case of a negated expression

Definition 3.11. Let P be a program and let M be an interpretation of P. Then M will be called a *model* of P iff for every rule $\mathsf{p}\,\mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$ in P and for every state s

$$\bigwedge\{[\![\mathsf{L}_1]\!]_s(M), \cdots, [\![\mathsf{L}_m]\!]_s(M)\} \leq_o [\![\mathsf{p}\,\mathsf{V}_1 \cdots \mathsf{V}_n]\!]_s(M)$$

3.2.2 Monotone-antimonotone and antimonotone-monotone functions

Before proceeding with a way to find the minimal WFS model of a program we need to present an alternative way to view Fitting-monotonic functions. Specifically, it is shown that every Fitting-monotonic function can be equivalently represented as a pair of functions (f_1, f_2), where f_1 is monotone-antimonotone, f_2 is antimonotone-monotone and $f_1 \leq f_2$. This section is required since the definition of the consequence operator we need is constructed in such a way so as to operate on pairs of such functions instead of their Fitting-monotonic function counterpart (see [2]).

Firstly, at the base one can view the truth values as pairs of truth values of the 2-valued model where *true* corresponds to $(\mathit{true}, \mathit{true})$, *false* corresponds to $(\mathit{false}, \mathit{false})$, and *undef* corresponds to $(\mathit{false}, \mathit{true})$. A function f can also equivalently be seen as a function f' that returns pairs. We can then break f' into two components f_1 and f_2 where f_1 returns the first element of the pair that f' returns while f_2 returns the second. The input of the function is also changed to be a pair of functions if the argument is of higher order. The monotone-antimonotone and antimonotone-monotone requirements ensure that the pair (f_1, f_2) retains the property of Fitting-monotonicity of the original function f. These ideas can be generalized to arbitrary types. The formal details of this equivalence are described below.

Definition 3.12. Let L_1, L_2 be sets and let $\leq$ be a partial order on $L_1 \cup L_2$. We define: $L_1 \otimes L_2 = \{(x, y) \in L_1 \times L_2 : x \leq y\}$.

Definition 3.13. Let L_1, L_2 be sets and let $\leq$ be a partial order on $L_1 \cup L_2$. Also, let $(A, \leq_A)$ be a partially ordered set. A function $f : (L_1 \otimes L_2) \rightarrow A$ will be called *monotone-antimonotone* (respectively *antimonotone-monotone*) if for all $(x, y), (x', y') \in L_1 \otimes L_2$ with $x \leq x'$ and $y' \leq y$, it holds that $f(x, y) \leq_A f(x', y')$ (respectively $f(x', y') \leq_A f(x, y)$). We denote by $[(L_1 \otimes L_2) \overset{\mathsf{ma}}{\rightarrow} A]$ the set of functions that are monotone-antimonotone and by $[(L_1 \otimes L_2) \overset{\mathsf{am}}{\rightarrow} A]$ those that are antimonotone-monotone.

With this new view, we will interpret the predicate types of our language as pairs of *monotone-antimonotone* and *antimonotone-monotone* functions defined recursively as follows.

Definition 3.14. Let D be a nonempty set. For every type τ we define the monotone-antimonotone and the antimonotone-monotone meanings of the elements of type τ with respect to D, denoted respectively by $[\![\rho]\!]_D^{\mathsf{ma}}$ and $[\![\rho]\!]_D^{\mathsf{am}}$. At the same time we define a partial order $\leq_\tau$ between the elements of $[\![\rho]\!]_D^{\mathsf{ma}} \cup [\![\rho]\!]_D^{\mathsf{am}}$.

- $[\![o]\!]_D^{\mathsf{ma}} = [\![o]\!]_D^{\mathsf{am}} = \{\mathit{false}, \mathit{true}\}$. The partial order $\leq_o$ is the usual one induced by the ordering *false* $\leq_o$ *true*.
- $[\![\iota]\!]^{\mathsf{ma}} = [\![\iota]\!]^{\mathsf{am}} = D$. The partial order $\leq_\iota$ is defined as $d \leq_\iota d$, for all $d \in D$.
- $[\![\iota \to \pi]\!]_D^{\mathsf{ma}} = D \to [\![\pi]\!]_D^{\mathsf{ma}}$ and $[\![\iota \to \pi]\!]_D^{\mathsf{am}} = D \to [\![\pi]\!]_D^{\mathsf{am}}$. The partial order $\leq_{\iota\to\pi}$ is defined as follows: for all $f, g \in [\![\iota \to \pi]\!]_D^{\mathsf{ma}} \cup [\![\iota \to \pi]\!]_D^{\mathsf{am}}$, $f \leq_{\iota\to\pi} g$ iff $f(d) \leq_\pi g(d)$ for all $d \in D$.
- $[\![\pi_1 \to \pi_2]\!]_D^{\mathsf{ma}} = [([\![\pi_1]\!]_D^{\mathsf{ma}} \otimes [\![\pi_1]\!]_D^{\mathsf{am}}) \overset{\mathsf{ma}}{\to} [\![\pi_2]\!]_D^{\mathsf{ma}}]$, and $[\![\pi_1 \to \pi_2]\!]_D^{\mathsf{am}} = [([\![\pi_1]\!]_D^{\mathsf{ma}} \otimes [\![\pi_1]\!]_D^{\mathsf{am}}) \overset{\mathsf{am}}{\to} [\![\pi_2]\!]_D^{\mathsf{am}}]$. The relation $\leq_{\pi_1\to\pi_2}$ is the partial order defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]_D^{\mathsf{ma}} \cup [\![\pi_1 \to \pi_2]\!]_D^{\mathsf{am}}$, $f \leq_{\pi_1\to\pi_2} g$ iff $f(d_1, d_2) \leq_{\pi_2} g(d_1, d_2)$ for all $(d_1, d_2) \in [\![\pi_1]\!]_D^{\mathsf{ma}} \otimes [\![\pi_1]\!]_D^{\mathsf{am}}$.

For every π, the bottom and top elements of $[\![\pi]\!]_D^{\mathsf{ma}}$ and $[\![\pi]\!]_D^{\mathsf{am}}$ can be defined in the obvious way. We present the following propositions where proofs for them can be found in [2].

Proposition 3.2 ([2]). *Let D be a nonempty set. For every predicate type π, $([\![\pi]\!]_D^{\mathsf{ma}}, \leq_\pi$) and $([\![\pi]\!]_D^{\mathsf{am}}, \leq_\pi)$ are complete lattices.*

We extend, in a pointwise way, our orderings to apply to pairs.

Definition 3.15. Let D be a nonempty set and let π be a predicate type. We define the relations $\leq_\pi$ and $\preceq_\pi$, so that for all $(x, y), (x', y') \in [\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$:

- $(x, y) \leq_\pi (x', y')$ iff $x \leq_\pi x'$ and $y \leq_\pi y'$.
- $(x, y) \preceq_\pi (x', y')$ iff $x \leq_\pi x'$ and $y' \leq_\pi y$.

Proposition 3.3 ([2]). *Let D be a nonempty set. For each predicate type π, $[\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$ is a complete lattice with respect to $\leq_\pi$ and a chain-complete poset with respect to $\preceq_\pi$.*

We will denote the *first* and *second* selection functions on pairs with the more compact notation $[\cdot]_1$ and $[\cdot]_2$: given any pair (x, y), it is $[(x, y)]_1 = x$ and $[(x, y)]_2 = y$. Then we define functions that move us from one representation to the other.

Definition 3.16. Let D be a nonempty set. For every predicate type π, we define recursively the functions $\tau_\pi : [\![\pi]\!]_D \to ([\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}})$ and $\tau_\pi^{-1} : ([\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}) \to [\![\pi]\!]_D$, as follows.

- $\tau_o(\mathit{false}) = (\mathit{false}, \mathit{false})$, $\tau_o(\mathit{true}) = (\mathit{true}, \mathit{true})$, $\tau_o(\mathit{undef}) = (\mathit{false}, \mathit{true})$
- $\tau_{\iota\to\pi}(f) = (\lambda d.[\tau_\pi(f(d))]_1, \lambda d.[\tau_\pi(f(d))]_2)$
- $\tau_{\pi_1\to\pi_2}(f) = (\lambda(d_1, d_2).[\tau_{\pi_2}(f(\tau_{\pi_1}^{-1}(d_1, d_2)))]_1, \lambda(d_1, d_2).[\tau_{\pi_2}(f(\tau_{\pi_1}^{-1}(d_1, d_2)))]_2)$

and

- $\tau_o^{-1}(\mathit{false}, \mathit{false}) = \mathit{false}$, $\tau_o^{-1}(\mathit{true}, \mathit{true}) = \mathit{true}$, $\tau_o^{-1}(\mathit{false}, \mathit{true}) = \mathit{undef}$
- $\tau_{\iota\to\pi}^{-1}(f_1, f_2) = \lambda d.\tau_\pi^{-1}(f_1(d), f_2(d))$
- $\tau_{\pi_1\to\pi_2}^{-1}(f_1, f_2) = \lambda d.\tau_{\pi_2}^{-1}(f_1(\tau_{\pi_1}(d)), f_2(\tau_{\pi_1}(d)))$.

We can now establish the bijection between $[\![\pi]\!]_D$ and $[\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$ with the following propositions. Again proofs for these can be found in [2].

Proposition 3.4 ([2]). *Let D be a nonempty set and let π be a predicate type. Then, for every $f, g \in [\![\pi]\!]_D$ and for every $(f_1, f_2), (g_1, g_2) \in [\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$, the following statements hold:*

1. $\tau_\pi(f) \in ([\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}})$ *and* $\tau_\pi^{-1}(f_1, f_2) \in [\![\pi]\!]_D$.
2. *If* $f \preceq_\pi g$ *then* $\tau_\pi(f) \preceq_\pi \tau_\pi(g)$.
3. *If* $f \leq_\pi g$ *then* $\tau_\pi(f) \leq_\pi \tau_\pi(g)$.
4. *If* $(f_1, f_2) \preceq_\pi (g_1, g_2)$ *then* $\tau_\pi^{-1}(f_1, f_2) \preceq_\pi \tau_\pi^{-1}(g_1, g_2)$.
5. *If* $(f_1, f_2) \leq_\pi (g_1, g_2)$ *then* $\tau_\pi^{-1}(f_1, f_2) \leq_\pi \tau_\pi^{-1}(g_1, g_2)$.

Proposition 3.5 ([2]). *Let D be a nonempty set and let π be a predicate type. Then, for every $f \in [\![\pi]\!]_D$, $\tau_\pi^{-1}(\tau_\pi(f)) = f$, and for every $(f_1, f_2) \in [\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$, $\tau_\pi(\tau_\pi^{-1}(f_1, f_2)) = (f_1, f_2)$.*

3.2.3 Well-Founded Semantics

We will now fully define what a Herbrand interpretation is for the language and how the bijection of Fitting monotone elements and pairs monotone-antimonotone and antimonotone-monotone functions extends to interpretations. Every program has a distinguished *minimal Herbrand model* which can be obtained by an iterative procedure we will present here by defining a new immediate consequence operator. The results are drawn from [2] which extends the approximation fix-point theory found in [6].

Definition 3.17. Let P be a program. The Herbrand universe U_{P} of P is the set of all the individual constants that appear in the program.

No function symbols are allowed in our language.

Definition 3.18. A (three-valued) *Herbrand interpretation* $\mathcal{I}$ of a program P is an interpretation such that:

1. the domain of $\mathcal{I}$ is the Herbrand universe U_{P} of P;
2. for every individual constant c of P, $\mathcal{I}(\mathsf{c}) = \mathsf{c}$;
3. for every predicate constant $\mathsf{p} : \pi$ of P, $\mathcal{I}(\mathsf{p}) \in [\![\pi]\!]_{U_{\mathsf{P}}}$;

We denote the set of all three-valued Herbrand interpretations of a program P by $\mathcal{H}_{\mathsf{P}}$. A *Herbrand state* of P is a state whose underlying domain is U_{P}. A *Herbrand model* of P is a Herbrand interpretation that is a model of P.

We extend the truth and the information orderings to Herbrand interpretations:

Definition 3.19. Let P be a program. We define the partial orders $\leq$ and $\preceq$ on $\mathcal{H}_{\mathsf{P}}$ as follows: for all $\mathcal{I}, \mathcal{J} \in \mathcal{H}_{\mathsf{P}}$, $\mathcal{I} \leq \mathcal{J}$ (respectively, $\mathcal{I} \preceq \mathcal{J}$) iff for every predicate type π and for every predicate constant $\mathsf{p} : \pi$ of P, $\mathcal{I}(\mathsf{p}) \leq_\pi \mathcal{J}(\mathsf{p})$ (respectively, $\mathcal{I}(\mathsf{p}) \preceq_\pi \mathcal{J}(\mathsf{p})$).

We will also need the following propositions which we present without proof here.

Proposition 3.6 ([2]). *Let P be a program. Then, $(\mathcal{H}_{\mathsf{P}}, \leq)$ is a complete lattice and $(\mathcal{H}_{\mathsf{P}}, \preceq)$ is a chain complete poset.*

Proposition 3.7 ([2]). *Let P be a program, let $\mathcal{I}, \mathcal{J} \in \mathcal{H}_{\mathsf{P}}$, and let s be a Herbrand state of P. For every expression E, if $\mathcal{I} \preceq \mathcal{J}$ then $[\![\mathsf{E}]\!]_s(\mathcal{I}) \preceq [\![\mathsf{E}]\!]_s(\mathcal{J})$.*

Now we will use the bijection established in section 3.2.1 to redefine interpretations this way. More specifically, every three-valued Herbrand interpretation $\mathcal{I}$ of a program P can be mapped by (an extension of) τ **to a pair of interpretations** (I, J) such that:

- for every individual constant c of P, $I(\mathsf{c}) = J(\mathsf{c}) = \mathsf{c}$;
- for every predicate constant $\mathsf{p} : \pi$ of P, $I(\mathsf{p}) \in [\![\pi]\!]^{\mathsf{ma}}_{U_{\mathsf{P}}}$ and $J(\mathsf{p}) \in [\![\pi]\!]^{\mathsf{am}}_{U_{\mathsf{P}}}$;

We will denote by $\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}}$ the set of functions of the former type and by $\mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$ those of the latter type. We can define a partial order $\leq$ on $\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \cup \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$. Similarly we can define partial orders $\leq$ and $\preceq$ on $\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$.

Proposition 3.8 ([2]). *Let P be a program. Then, $(\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}}, \leq)$ and $(\mathcal{H}^{\mathsf{am}}_{\mathsf{P}}, \leq)$ are complete lattices having the same $\perp$ and $\top$ elements. Moreover, $(\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}, \leq)$ is a complete lattice and $(\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}, \preceq)$ is a chain-complete poset.*

The bijection between $\mathcal{H}_{\mathsf{P}}$ and $\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$ can be explained more formally as follows. Given $\mathcal{I} \in \mathcal{H}_{\mathsf{P}}$, we define $\tau(\mathcal{I}) = (I, J)$, where for every predicate constant $\mathsf{p} : \pi$ it holds $I(\mathsf{p}) = [\tau_\pi(\mathcal{I}(\mathsf{p}))]_1$ and $J(\mathsf{p}) = [\tau_\pi(\mathcal{I}(\mathsf{p}))]_2$. Conversely, given a pair $(I, J) \in \mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$, we define the three-valued Herbrand interpretation $\mathcal{I}$ as follows: $\mathcal{I}(\mathsf{p}) = \tau_\pi^{-1}(I(\mathsf{p}), J(\mathsf{p}))$.

For (mostly computational) convenience in later algorithms, we will give an equivalent definition for the semantics of expressions by using only the monotone-antimonotone and the antimonotone-monotone meanings of the elements for our types directly and without the use of τ function.

Definition 3.20. A ***pair*** Herbrand state s over U_{P} is a function that assigns to each argument variable V of type ρ to an element $s(\mathsf{V}) \in [\![\pi]\!]^{\mathsf{ma}}_{U_{\mathsf{P}}} \otimes [\![\pi]\!]^{\mathsf{am}}_{U_{\mathsf{P}}}$.

We define: $(\mathit{true}, \mathit{true})^{-1} = (\mathit{false}, \mathit{false})$, $(\mathit{false}, \mathit{true})^{-1} = (\mathit{false}, \mathit{true})$ and finally $(\mathit{false}, \mathit{false})^{-1} = (\mathit{true}, \mathit{true})$. For the semantics of the expressions we have:

Definition 3.21. Let $(I, J) \in \mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$ be a pair of interpretations such that $\mathcal{I} = \tau^{-1}(I, J)$ where $\mathcal{I}$ is a three-valued Herbrand interpretation of P. Also, let s be a two-valued state over U_{P}. The semantics of expressions then are defined recursively as:

1. $[\![\mathsf{c}]\!]_s(I, J) = \mathsf{c}$, for every individual constant c
2. $[\![\mathsf{p}]\!]_s(I, J) = (I(\mathsf{p}), J(\mathsf{p}))$, for every predicate constant
3. $[\![\mathsf{V}]\!]_s(I, J) = s(\mathsf{V})$, for every variable
4. $[\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(I, J) = ([[\![\mathsf{E}_1]\!]_s(I, J)]_1([\![\mathsf{E}_2]\!]_s(I, J)), [[\![\mathsf{E}_1]\!]_s(I, J)]_2([\![\mathsf{E}_2]\!]_s(I, J)))$
5. $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_s(I, J) = \begin{cases} (\mathit{true}, \mathit{true}), & \text{if } [\![\mathsf{E}_1]\!]_s(I, J) = [\![\mathsf{E}_2]\!]_s(I, J) \\ (\mathit{false}, \mathit{false}), & \text{otherwise} \end{cases}$
6. $[\![(\sim\mathsf{E})]\!]_s(I, J) = ([\![\mathsf{E}]\!]_s(I, J))^{-1}$ for the negation of an expression

We will use the brackets "$[\![$", and"$]\!]$" when we refer to evaluation of an expression under both three-valued and two-valued pair interpretations. It is always clear from the context which one we refer to.

We have the following lemma that binds the two equivalent semantic views for expressions.

Lemma 3.9. *Let* P *be a program. Let* s *be a two-valued state over* U_{P} *and* s' *be a three-valued state over* U_{P} *such as for every variable* V *it holds:*

$$s(\mathsf{V}) = \tau(s'(\mathsf{V}))$$

Then it is:

$$[\![\mathsf{E}]\!]_s(I,J) = \tau([\![\mathsf{E}]\!]_{s'}\tau^{-1}(I,J))$$

Proof. The proof of this lemma is an induction on the way expressions are constructed. In the case of program constants and variables, it follows directly from the definitions. For the case of equality and negation, it is also trivial to show. In the case of the application rule, we will have to use the fact that

$$\tau(f(g)) = ([\tau(f)]_1(\tau(g)), [\tau(f)]_2(\tau(g))) \qquad (1)$$

which can be derived from the third bullet of the definition 3.16. Specifically, we have that

$$\begin{aligned}
\tau([\![(\mathsf{E}_1\mathsf{E}_2)]\!]_{s'}\tau^{-1}(I,J)) &\overset{def\ 3.10}{=} \tau\left([\![(\mathsf{E}_1)]\!]_{s'}\tau^{-1}(I,J)([\![\mathsf{E}_1]\!]_{s'}\tau^{-1}(I,J))\right)\\
&\overset{(1)}{=} \left[\tau([\![\mathsf{E}_1]\!]_{s'}\tau^{-1}(I,J))\right]_1\left(\tau([\![\mathsf{E}_1]\!]_{s'}\tau^{-1}(I,J))\right),\\
&\qquad \left[\tau([\![\mathsf{E}_1]\!]_{s'}\tau^{-1}(I,J))\right]_2\left(\tau([\![\mathsf{E}_1]\!]_{s'}\tau^{-1}(I,J))\right)\\
&\overset{i.h.}{=} [[\![\mathsf{E}_1]\!]_s(I,J)]_1([\![\mathsf{E}_2]\!]_s(I,J)),\\
&\qquad [[\![\mathsf{E}_1]\!]_s(I,J)]_2([\![\mathsf{E}_2]\!]_s(I,J))\\
&\overset{def\ 3.21}{=} [\![(\mathsf{E}_1\mathsf{E}_2)]\!]_s(I,J)
\end{aligned}$$

□

Notice that the function τ is an isomorphism between the set of Fitting-monotonic functions on a domain D and the set of valid pairs of monotone-antimonotone, antimonotone-monotone functions defined on the same domain D. In this work, in later chapters, we work almost exclusively with the two-valued domain and function pairs.

We now define the three-valued and two-valued immediate consequence operators:

Definition 3.22. Let P be a program. The *three-valued immediate consequence operator* $\Psi_{\mathsf{P}} : \mathcal{H}_{\mathsf{P}} \to \mathcal{H}_{\mathsf{P}}$ of P is defined for every predicate constant $\mathsf{p} : \rho_1 \to \cdots \to \rho_n \to o$ and $d_i \in [\![\rho_i]\!]$ as:

$$\Psi_{\mathsf{P}}(\mathcal{I})(\mathsf{p})d_1\cdots d_n =$$

$$\bigvee\{\bigwedge[\![\mathsf{L}_i]\!]_{s[\mathsf{V}_1/d_1,\ldots,\mathsf{V}_n/d_n]}(\mathcal{I}) \mid (\mathsf{p}\,\mathsf{V}_1\cdots\mathsf{V}_n \leftarrow \mathsf{L}_1\wedge\cdots\wedge\mathsf{L}_m) \in \mathsf{P},\ s \text{ a Herbrand state}\}$$

Definition 3.23. Let P be a program. The *two-valued immediate consequence operator* $T_{\mathsf{P}} : (\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}) \to (\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}})$ of P is defined as:

$$T_{\mathsf{P}}(I,J) = \tau(\Psi_{\mathsf{P}}(\tau^{-1}(I,J)))$$

We will also give an equivalent definition for the two-valued consequence operator that is easier to handle since it does not depend on Ψ_{P}. The equivalence of the two definitions is a consequence of Lemma 3.9.

Definition 3.24. [Alternative] Let P be a program. The *two-valued immediate consequence operator* $T_{\mathsf{P}} : (\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}) \to (\mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}})$ of P is defined for every predicate constant $\mathsf{p} : \rho_1 \to \cdots \to \rho_n \to o$ and $d_i \in [\![\rho_i]\!]^{\mathsf{ma}} \otimes [\![\rho_i]\!]^{\mathsf{am}}$ as:

$$T_{\mathsf{P}}(I,J)(\mathsf{p})d_1 \cdots d_n =$$

$$\bigvee\{\bigwedge [\![\mathsf{L}_i]\!]_{s[\mathsf{V}_1/d_1,\dots,\mathsf{V}_n/d_n]}(I,J) \mid (\mathsf{p}\,\mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m) \in \mathsf{P},\ s \text{ a pair Herbrand state}\}$$

It follows that T_{P} is well-defined. Moreover, it is Fitting-monotonic as the following lemma demonstrates:

Lemma 3.10 ([2])**.** *Let* P *be a program and let* $(I_1, J_1), (I_2, J_2) \in \mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$. *If* $(I_1, J_1) \preceq (I_2, J_2)$ *then* $T_{\mathsf{P}}(I_1, J_1) \preceq T_{\mathsf{P}}(I_2, J_2)$.

The T_{P} operator is used to construct the well-founded model of program P but in a bit more involved process than what we used in the positive case.

Definition 3.25. Let P be a program and let $(I,J) \in \mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$. Assume that $(I,J) \preceq T_{\mathsf{P}}(I,J)$. We define $I^{\uparrow} = \mathit{lfp}([T_{\mathsf{P}}(I,\cdot)]_2)$ and $J^{\downarrow} = \mathit{lfp}([T_{\mathsf{P}}(\cdot,J)]_1)$, where by $T_{\mathsf{P}}(\cdot,J)$ we denote the function $f(x) = T_{\mathsf{P}}(x,J)$ and by $T_{\mathsf{P}}(I,\cdot)$ the function $g(x) = T_{\mathsf{P}}(I,x)$.

It can be shown that $I^{\uparrow}$ and $J^{\downarrow}$ are well-defined, and this is due to the crucial assumption $(I,J) \preceq T_{\mathsf{P}}(I,J)$. Finally, we need to define one more operator, namely the *stable revision operator* which we will denote as $\mathcal{C}_{T_{\mathsf{P}}}$.

Definition 3.26. Let P be a program. We define the function $\mathcal{C}_{T_{\mathsf{P}}}$ which for every pair $(I,J) \in \mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$ with $(I,J) \preceq T_{\mathsf{P}}(I,J)$, returns the pair $(J^{\downarrow}, I^{\uparrow})$:

$$\mathcal{C}_{T_{\mathsf{P}}}(I,J) = (J^{\downarrow}, I^{\uparrow}) = (\mathit{lfp}([T_{\mathsf{P}}(\cdot,J)]_1), \mathit{lfp}([T_{\mathsf{P}}(I,\cdot)]_2))$$

The function $\mathcal{C}_{T_{\mathsf{P}}}$ will be called the *stable revision operator* for T_{P}.

The following theorem gives us the iterative process we seek:

Theorem 3.11 ([2])**.** *Let* P *be a program. We define the following sequence of pairs of interpretations:*

$$\begin{array}{lcl} (I_0, J_0) & = & (\bot, \top) \\ (I_{\lambda+1}, J_{\lambda+1}) & = & \mathcal{C}_{T_{\mathsf{P}}}(I_\lambda, J_\lambda) \end{array}$$

Then, the above sequence of pairs of interpretations is well-defined. Moreover, there exists a a natural number δ *such that* $(I_\delta, J_\delta) = \mathcal{C}_{T_{\mathsf{P}}}(I_\delta, J_\delta)$ *and* $(I_\delta, J_\delta) \in \mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$.

In the following, we will denote with $\mathcal{M}_{\mathsf{P}}$ the interpretation $\tau^{-1}(I_\delta, J_\delta)$. The following two lemmas demonstrate that the pre-fixpoints of T_{P} correspond exactly to the three-valued models of P.

Lemma 3.12 ([2])**.** *Let* P *be a program. If* $(I,J) \in \mathcal{H}^{\mathsf{ma}}_{\mathsf{P}} \otimes \mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$ *is a pre-fixpoint of* T_{P} *then* $\tau^{-1}(I,J)$ *is a model of* P.

Lemma 3.13 ([2])**.** *Let* $\mathcal{M} \in \mathcal{H}_{\mathsf{P}}$ *be a model of* P. *Then,* $\tau(\mathcal{M})$ *is a pre-fixpoint of* T_{P}.

Finally, we have the following two lemmas:

Theorem 3.14 ([2]). *Let* P *be a program. Then,* $\mathcal{M}_{\mathsf{P}}$ *is a* $\leq$*-minimal model of* P.

Theorem 3.15 ([2]). *For every propositional program* P, $\mathcal{M}_{\mathsf{P}}$ *coincides with the well-founded model of* P.

To argue about complexity we also need to give a way to calculate both *lfp*$([T_{\mathsf{P}}(\cdot, J_n)]_1)$ and *lfp*$([T_{\mathsf{P}}(I_n, \cdot)]_2)$ through an iterative process.
For *lfp*$([T_{\mathsf{P}}(\cdot, J_n)]_1)$ it is the least upper bound of the following sequence:

$$\begin{array}{rcl} I^1_{n+1} & = & [T_{\mathsf{P}}(\bot, J_n)]_1 \\ I^2_{n+1} & = & [T_{\mathsf{P}}(I^1_{n+1}, J_n)]_1 \\ & \dots & \\ I^{\alpha+1}_{n+1} & = & [T_{\mathsf{P}}(I^{\alpha}_{n+1}, J_n)]_1 \\ & \dots & \end{array}$$

and for *lfp*$([T_{\mathsf{P}}(I_n, \cdot)]_2)$ the least upper bound of the sequence:

$$\begin{array}{rcl} J^1_{n+1} & = & [T_{\mathsf{P}}(I_n, I^*_n)]_2 \\ J^2_{n+1} & = & [T_{\mathsf{P}}(I_n, J^1_{n+1})]_2 \\ & \dots & \\ J^{\alpha+1}_{n+1} & = & [T_{\mathsf{P}}(I_n, J^{\alpha}_{n+1})]_2 \\ & \dots & \end{array}$$

where I^*_n is the least interpretation in $\mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$ such that $I_n \leq I^*_n$, namely the bottom antimonotone-monotone element of the interval $[I_n, \top]$.

This is the core loop of the algorithm we will describe that produces the minimal WFS model we will present in a following chapter. The sequence of the interpretations in each loop are non-decreasing since it holds $I^{k+1}_{n+1} \geq I^k_{n+1}$ and $J^{k+1}_{n+1} \geq J^k_{n+1}$ by the properties of the operator. With the former assumptions, the sequences stop at finite steps for our function-less language operating on finite domains of constants. They stop in the sense that there will be a natural number N big enough such that $I^{N+1}_{n+1} = I^N_{n+1}$ and $J^{N+1}_{n+1} = J^N_{n+1}$ which means that the fix-point element belongs to the sequences. To verify this one has to notice that since the base domain for type individual is finite since it is the constants of the program and the input database, for every predicate type that we have in the program the total number of elements we can create of such type has to be finite as well. If no constant predicate of the program changes in one iteration of an inner loop or after a step of the outer loop this signals convergence. Therefore, the number of iterations of the nested loops must be finite.

3.3 An equivalence of WFS and Positive semantics on positive programs

The following proposition restricts any ground atom from getting the value *undef* in the WFS model when negation is not used in the program and also forces it to have the same evaluation as if it was calculated under classic positive semantics. Notice however that the Fitting-monotone functions that the higher-order predicate constants are assigned to in the three-valued model are defined over a different domain than that of the monotone functions of the two-valued model under positive semantics.

Proposition 3.16. *Let* P *be a positive program of higher-order Datalog. Let* $\mathcal{M}_{wfs}$ *be the model under Well-founded semantics and* $\mathcal{M}_c$ *be the model under classic positive semantics. Then for every ground atom* g *it is* $\mathcal{M}_{wfs}(\mathsf{g}) = \mathcal{M}_c(\mathsf{g})$.

We will only prove formally the proposition for any language fragment $\mathcal{H}_k^{\exists}$, $k \geq 1$ since it is enough for the purpose of this thesis [1], essentially ignoring the possibility of having partial application in the body of the rules. We will denote with $[\![\rho]\!]^{wfs}$ the set of elements for type ρ under Well-founded model of semantics and $[\![\rho]\!]^{c}$ the set of elements under classic positive semantics. First, we consider the following definitions.

Definition 3.27. A $d \in [\![\rho]\!]^{wfs}$ will be called *well-behaved* iff

- $\rho = o$
- $\rho = \iota$
- $\rho = \rho_1 \to \rho_2 \to \ldots \to \rho_t \to o$ and for any set of indices $S \subseteq \{1, \ldots, t\}$ and for any two elements $\bar{d} = (d_1, \ldots, d_t)$, $\bar{d}' = (d'_1, \ldots, d'_t)$ such that:
  - if $i \in S$ and ρ_i has a top element for the ordering $\leq$, then d_i is that top element and d'_i is an arbitrary element in $[\![\rho_i]\!]^{wfs}$.
  - if $i \notin S$ or ρ_i does not have a top element then $d_i = d'_i$ and $d_i, d'_i \in [\![\rho_i]\!]^{wfs}$ are well-behaved.

  Then it must hold that $d(\bar{d}) \geq d(\bar{d}')$.

Intuitively consider S to be the set of indices of some non-fixed arguments and the rest of the fixed arguments are all well-behaved elements. Then the maximum truth value for the head element when fully applied is taken by choosing every non-fixed argument in S to be the top element of that type. This is analogous to the monotonicity of the positive semantics case. Only that there, it holds for the whole domain for each type and we can always swap an argument with a top element without reducing the output value. In this case, we just require this to hold while the fixed arguments are all well-behaved.

Definition 3.28. A $d^w \in [\![\rho]\!]^{wfs}$ will be called an "I-extension" of a $d^c \in [\![\rho]\!]^{c}$ iff

- $\rho = \iota$ and $d^w = d^c$
- $\rho = o$ and $[d]_1 = d^c$
- $\rho = \rho_1 \to \rho_2$, d^w is *well-behaved* and $\forall d^w_{in}, d^c_{in}$ where $d^w_{in} \in [\![\rho_1]\!]^{wfs}$, $d^c_{in} \in [\![\rho_1]\!]^{c}$ and d^w_{in} is an I-extension of d^c_{in} we have that $d^w(d^w_{in})$ is an I-extension of $d^c(d^c_{in})$.

Furthermore, we will call a pair interpretation (I, J) an I-extension of a positive interpretation $\mathcal{I}$ iff for every constant predicate p, $(I, J)(\mathsf{p})$ is an I-extension of $\mathcal{I}(\mathsf{p})$.

Lemma 3.17. *Let* P *be a positive higher-order Datalog program in* $\mathcal{H}_k^{\exists}$, $k \geq 1$ *and consider the following sequence that produces the WFS model of the program.*

$$\begin{array}{rcl} (I_0, J_0) & = & (\bot, \top) \\ (I_1, J_1) & = & \mathcal{C}_{T_{\mathsf{P}}}(I_0, J_0) \\ \ldots & & \end{array}$$

Let $\mathcal{I}$ *be the minimum model of the program* P *under positive semantics. Then* (I_1, J_0) *is an* I*-extension of* $\mathcal{I}$.

[1]We use this result for the fragments of chapter 6.

Proof. Let $I_1^0, I_1^1, I_1^2, \ldots$ be the sequence that converges to I_1 by the algorithm of producing the WFS model.

$$\begin{array}{rcl} I_1^1 & = & [T_{\mathsf{P}}(I_1^0 = \perp, J_0)]_1 \\ I_1^2 & = & [T_{\mathsf{P}}(I_1^1, J_0)]_1 \\ & \ldots & \end{array}$$

Also let $\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2, \ldots$ be the convergence sequence of minimum model $\mathcal{I}$ under positive semantics.

$$\begin{array}{rcl} \mathcal{I}_1 & = & T^c_{\mathsf{P}}(\mathcal{I}_0) \\ \mathcal{I}_2 & = & T^c_{\mathsf{P}}(\mathcal{I}_1) \\ & \ldots & \end{array}$$

where we denote as T^c_{P} the immediate sequence operator for the positive semantics to avoid confusion. We can show that for any $k \geq 0$, (I_1^k, J_0) is an I-extension of $\mathcal{I}_k$. We will use induction to prove it. For the base case, it is easy to show that (I_1^0, J_0) is an I-extension of $\mathcal{I}_0$.

Assume that it holds for $k \geq 0$. Then we have that (I_1^k, J_0) is *well-behaved*. We show that (I_1^{k+1}, J_0) is also *well-behaved*. For any constant predicate p pick an arbitrary $S \subseteq \{1, \ldots, t\}$ and any $\bar{d}, \bar{d'}$ that fit the requirements of the definition 3.27. We compare the two output values of $(I_1^{k+1})(\mathsf{p})$ at $\bar{d}$ and $\bar{d'}$. Consider only one rule in the program with p in its head and no existential zero-order variables in the body for an easier analysis [2]. The generalisation to many rules is straightforward. Let $\mathsf{V}_1, \ldots, \mathsf{V}_t$ be the set of formal variables and $\mathsf{V}_{t+1}, \ldots, \mathsf{V}_{t'}$ the higher-order existential variables. Then we have:

$$I_1^{k+1}(\mathsf{p})\bar{d} = \bigvee_s \left\{ \bigwedge_l [\![E_l]\!]_s(I_1^k, J_0) \mid s(\mathsf{V}_i) = d_i, i \leq t \right\} \tag{1}$$

$$I_1^{k+1}(\mathsf{p})\bar{d'} = \bigvee_s \left\{ \bigwedge_l [\![E_l]\!]_s(I_1^k, J_0) \mid s(\mathsf{V}_i) = d'_i, i \leq t \right\} \tag{2}$$

where each E_l is an atom and $\bigvee_s$ is taken over every s such that if $i \leq t$ it is $s(\mathsf{V}_i) = d_i$ in the first case and $s(\mathsf{V}_i) = d'_i$ in the second case. Let s_o be the optimal state which maximizes the right-hand side of (1). We get that

$$I^{k+1}(\mathsf{p})\bar{d} = \bigwedge_l [\![E_l]\!]_{s_o}(I_1^k, J_0)$$

Because every d_i as well as the interpretation (I_1^k, J_0) is well-behaved we can safely argue that for every $i \geq t$, $s_o(V_i)$ can be set to the top element of the corresponding type of the existential variable V_i. Also let s'_o be the state that maximizes the right-hand side of (2) so we have that:

$$I^{k+1}(\mathsf{p})\bar{d'} = \bigwedge_l [\![E_l]\!]_{s'_o}(I_1^k, J_0)$$

It is easy by case analysis to show that for every atom E_l it holds

$$[\![E_l]\!]_{s_o}(I_1^k, J_0) \geq [\![E_l]\!]_{s'_o}(I_1^k, J_0)$$

[2]Instantiate all zero-order existential variables and produce multiple rules out of one original rule thus producing an equivalent program.

which proves that $I^{k+1}(\mathsf{p})\bar{d} \geq I^{k+1}(\mathsf{p})\bar{d}'$ and therefore

$$(I^{k+1}, J_0)(p)\bar{d} \geq (I^{k+1}, J_0)(p)\bar{d}'$$

Since this holds for any $\bar{d}, \bar{d}'$ that fit the requirements of the definition 3.27 we get that $(I^{k+1}, J_0)(p)$ is *well-behaved*. Indeed we have:

- E_i =="$(\mathsf{V}_i \ldots)$" for $i \in S$ or $i > t$ then it trivially holds since $s_o(\mathsf{V}_i)$ is the top element.
- E_i =="$(\mathsf{V}_i\ arg_1\ arg_2 \ldots)$" for $i \notin S$ and $i \leq t$. Then $s_o(\mathsf{V}_i) = s'_o(\mathsf{V}_i) = d_i$ which is also a *well-behaved* element. It remains to notice that for every argument arg_j it is either $[\![arg_j]\!]_{s_o}(I_1^k, J_0) = [\![arg_j]\!]_{s'_o}(I_1^k, J_0)$ and is a *well-behaved* element or $[\![arg_j]\!]_{s_o}(I_1^k, J_0)$ is the top element.
- E_i =="$(\mathsf{q}_i \ldots)$" for a constant predicate q. Same as the previous case since $(I_1^k, J_0)(\mathsf{q})$ is well-behaved.

Now it is left to show that for any $\bar{d} = (d_1, \ldots, d_t)$ and $\bar{d}' = (d'_1, \ldots, d'_t)$ where each d_i is an I-extension of d'_i we have that:

$$[(I_1^{k+1}, J_0)(\mathsf{p})\bar{d}]_1 = \mathcal{I}_{k+1}(\mathsf{p})\bar{d}'$$

which implies that $(I_1^{k+1}, J_0)(\mathsf{p})$ is an I- extension of $\mathcal{I}_{k+1}(\mathsf{p})$. Assume again one rule per predicate and no existential zero-order variables. In positive semantics, by using the monotonicity condition, we can set every higher-order existential variable to the corresponding top element. Then we have:

$$\mathcal{I}_{k+1}(\mathsf{p})\bar{d}' = \bigwedge_l [\![\mathsf{E}_l]\!]_{s^c}(\mathcal{I}_k)$$

where $s^c(\mathsf{V}_i) = d'_i$ for $i \leq t$ and $s^c(\mathsf{V}_i) = \top^c_{\rho_i}$ for $i > t$.
Due to the *well-behavedness* condition we also have:

$$I^{k+1}(\mathsf{p})\bar{d} = \bigwedge_l [\![\mathsf{E}_l]\!]_s(I_1^k, J_0)$$

where where $s(\mathsf{V}_i) = d_i$ for $i \leq t$ and $s(\mathsf{V}_i) = \top_{\rho_i}$ for $i > t$ since if every d_i is an extension it is also well-behaved. Notice also that naturally the top element $\top_{\rho_i}$ is an I- extension of the top element $\top^c_{\rho_i}$. Finally, it must be that for each E_l

$$[\![\mathsf{E}_l]\!]_s(I_1^k, J_0) = [\![\mathsf{E}_l]\!]_{s^c}(\mathcal{I}_k)$$

and therefore

$$I_1^{k+1}(\mathsf{p})\bar{d} = \mathcal{I}_{k+1}(\mathsf{p})\bar{d}'$$

This fact follows by noticing by per case analysis that each expression evaluation $[\![\mathsf{E}_l]\!]_s(I_1^k, J_0)$ is an I- extension of the corresponding evaluation $[\![\mathsf{E}_l]\!]_{s^c}(\mathcal{I}_k)$. The critical step was the ability to swap every higher-order existential variable to their top element in the truth ordering in the calculation of $I^{k+1}(\mathsf{p})\bar{d}$. Finally, in the limit we get

$$[(I_1, J_0)(\mathsf{p})\bar{d}]_1 = \mathcal{I}(\mathsf{p})\bar{d}'$$

□

Now we proceed with the definition of J-extension.

Definition 3.29. A $d^w \in [\![\rho]\!]^{wfs}$ will be called an "J-extension" of a $d^c \in [\![\rho]\!]^c$ iff

- $\rho = \iota$ and $d^w = d^c$
- $\rho = o$ and $[d]_2 = d^c$
- $\rho = \rho_1 \to \rho_2$, d^w is *well-behaved* and $\forall d^w_{in}, d^c_{in}$ where $d^w_{in} \in [\![\rho_1]\!]^{wfs}$, $d^c_{in} \in [\![\rho_1]\!]^c$ and d^w_{in} an J-extension of d^c_{in} we have that $d^w(d^w_{in})$ is an J-extension of $d^c(d^c_{in})$.

Furthermore, we will call a pair interpretation (I, J) an J-extension of a positive interpretation $\mathcal{I}$ iff for every constant predicate p, $(I, J)(\mathsf{p})$ is an J-extension of $\mathcal{I}(\mathsf{p})$.

We have the corresponding Lemma to the previous one:

Lemma 3.18. *Let* P *be a positive higher-order Datalog program in* $\mathcal{H}^{\exists}_k$, $k \geq 1$ *and consider the sequence that produces the WFS model of the program.*

$$\begin{array}{rcl} (I_0, J_0) & = & (\bot, \top) \\ (I_1, J_1) & = & \mathcal{C}_{T_\mathsf{P}}(I_0, J_0) \\ \ldots & & \end{array}$$

Let $\mathcal{I}$ *be the model of* P *with positive semantics. Then* (I_0, J_1) *is an* J*-extension of* $\mathcal{I}$.

Proof. The proof is similar to Lemma 3.17. Notice that $J_1^0 = \bot = I_0$ which means that (I_0, J_1^0) is a J-extension of $\mathcal{I}_0$. □

Now we can show the main theorem of this section.

Theorem 3.19. *Let* P *be a positive program that belongs in* $\mathcal{H}^{\exists}_k$ *and let* $\mathcal{M}_{wfs} = (I, J)$ *be the model under Well-Founded-Semantics and* $\mathcal{M}_c = \mathcal{I}$ *be the model under classic positive semantics. Then for every ground atom* g, $\mathcal{M}_{wfs}(\mathsf{g}) = {}^3\mathcal{M}_c(\mathsf{g})$

Proof. Let g be a ground atom.

- By Lemma 3.17 we have that $[\![\mathsf{g}]\!](I_1, J_0)$ is an I- extension of $[\![\mathsf{g}]\!](\mathcal{I})$. Therefore, if $[\![\mathsf{g}]\!](\mathcal{I}) =$ *true* then $[[\![\mathsf{g}]\!](I_1, J_0)]_1 =$ *true* and by the Fitting-monotonicity condition $[[\![\mathsf{g}]\!](I, J)]_1 =$ *true*. This enforces $[\![\mathsf{g}]\!](I, J) = ($*true*, *true*$)$.
- By Lemma 3.18 we have that $[\![\mathsf{g}]\!](I_0, J_1)$ is a J- extension of $[\![\mathsf{g}]\!](\mathcal{I})$. Therefore, if $[\![\mathsf{g}]\!](\mathcal{I}) =$ *false* then $[[\![\mathsf{g}]\!](I_0, J_1)]_2 =$ *false* and by the Fitting-monotonicity condition $[[\![\mathsf{g}]\!](I, J)]_2 =$ *false*. This enforces $[\![\mathsf{g}]\!](I, J) = ($*false*,*false*$)$.

□

$^3\mathcal{M}_{wfs}(\mathsf{g})$ when viewed in the three-valued model is the same as the value of $\mathcal{M}_c(\mathsf{g})$.

3.4 Decision problems with Datalog

In this section, we will give a formal definition of deciding a language through Datalog. This will establish the context in which we will see the expressive capabilities of each fragment of higher-order Datalog we consider.

3.4.1 Relevant complexity classes

We will need the following family of functions [9]:

$$\begin{array}{rcl} \exp_0(x) & = & x \\ \exp_{n+1}(x) & = & 2^{\exp_n(x)} \end{array}$$

For all $k \geq 0$, the complexity class $k - \mathsf{EXPTIME}$ is defined as follows:

$$k - \mathsf{EXPTIME} = \bigcup_{r \in \mathbb{N}} \mathsf{TIME}(\exp_k(n^r))$$

For the limit of k going to ∞ the class is called `ELEMENTARY`. Notice also that $0 - \mathsf{EXPTIME}$ is PTIME.

3.4.2 Decision problems and propositional predicates

Let Σ be an alphabet and without loss of generality, we fix $\Sigma = \{a, b\}$. We will assume that finite strings over Σ will always be encoded by an input relation such as the following which corresponds to the string "abba".

```
input 0 a 1.
input 1 b 2.
input 2 b 3.
input 3 a end.
```

More generally for an input string of $n > 0$ length, we can derive an input relation as we did above. There will be two constants "`a`" and "`b`" and another n constants (presented as "`0`", "`1`" etc) as well as the constant "`end`". Notice how the input is given ordered. All constant individual elements apart from "`a`", "`b`", and "`end`" are ordered by an order implicitly given by the input relation. For the input of length $n = 0$, namely for the empty string, we use:

```
input 0 empty end.
```

where `empty` is a constant that denotes the empty string.

Given string $w \in \Sigma^*$, we will write $\mathcal{D}_w$ to denote the set of facts that represent w through the `input` relation. This encoding of input strings is usually also referred to as the *ordered database assumption*.

Next, we will need a way for Datalog to decide if an input of the mentioned form is accepted or not. We will assume that every program defines a propositional `accept` predicate which, intuitively, signals whether a particular input string is accepted by our program.

Definition 3.30. Let Σ be an alphabet. We will say that a Higher-Order Datalog program P *decides* a language $L \subseteq \Sigma^*$ if for any $w \in \Sigma^*$, $w \in L$ iff `accept` is true in the Well-Founded semantics model of $\mathsf{P} \cup \mathcal{D}_w$.

Definition 3.31. We will say that a fragment $\mathcal{Q}$ of Higher-Order Datalog captures the complexity class $\mathcal{C}$, if the set of formal languages decided by programs that belong in $\mathcal{Q}$ coincides with $\mathcal{C}$.

CHAPTER 4

DATALOG WITH PARTIAL APPLICATION

In this chapter, we study Datalog fragments where partial application is allowed. Namely, the fragments that belong to the first row of the table.

partial application	negation	h.o. existential variables	order 1	order 2	order 3	$\cdots$	order ∞
YES	X	X	PTIME	EXPTIME	2−EXPTIME	$\cdots$	ELEM.

This row essentially represents four fragment sets, depending on the selection of 'YES' and 'NO' for the X's. However, we will demonstrate that allowing partial application alone is sufficient to maintain the hierarchy, regardless of the other two choices. For any k ≥ 1, each of these four fragments is equivalent in expressiveness. To maintain brevity, we will not delve deeply into the separate analysis of these fragment sets since their distinctions ultimately do not hold significance. Toward the end of this chapter, we will provide a high-level description of a Turing machine (an algorithm) capable of performing the bottom-up calculation to find the WFS model of a program.

4.1 Theorems

First, we consider the following theorem.

Theorem 4.1 ([1]). *For every $k \geq 1$, the set of k-order positive Datalog programs captures $(k-1) - \mathsf{EXPTIME}$.*

Notice that the simulation of a Turing Machine that decides L in [1] is given by a datalog program that ultimately belongs to every fragment considered in this section. That is because neither negation nor existential variables are used. This observation about the simulation gives us the following corollary by reusing it for our purpose.

Corollary 4.2. *For any $k \geq 1$ each language fragment of $\mathcal{H}_k^{\lambda}$, $\mathcal{H}_k^{\lambda,\exists}$, $\mathcal{H}_k^{\lambda,\neg}$ and $\mathcal{H}_k^{\lambda,\exists,\neg}$ can express at least $(k-1) - \mathsf{EXPTIME}$.*

What remains in order to prove the expressiveness is to give an upper bound of it. That is by bounding the running time we need to implement such versions of Datalog. We have the following lemma.

Lemma 4.3. *Let* P *be a* k*-order Datalog program with program order* $k \geq 1$*. Then there exists a Turing machine that for any set of input database* D_{in}*, given* D_{in} *as encoded input* [1]*, it can calculate the WFS model* $\mathcal{M}_{\mathsf{P}\cup\mathsf{D}_{in}}$ *within time* $O(\exp_{k-1}(n^q))$*, where* n *is the size of the representation of* D_{in} *and* q *is a constant that depends only on the program* P.

Proof. The existence of said Turing Machine and an argument of its running time-complexity are shown in section 4.2 of this chapter. This machine is tailored to perform the bottom-up calculation under WFS in contrast to the machine given in [1] which works for positive semantics and a language without the negation operator. □

This Lemma gives us the following corollary result.

Corollary 4.4. *Let* P *be a higher-order Datalog program with program order* $k \geq 1$ *that decides* L*. There exists a Turing machine that decides* L *and runs in time* $O(\exp_{k-1}(n^q))$*, where* n *is the length of the input string and* q *is a constant that depends only on* P.

Proof. This stems from 4.3 as follows. Let P be a program higher-order Datalog program that decides L. There exists a Turing machine that for every string w and corresponding input relation D_w finds the model of $\mathsf{P} \cup \mathsf{D}_w$. Furthermore, the machine does that in time $O(\exp_{k-1}(n^q))$ where $n = |\mathsf{D}_w|$ and by the assumed form of D_w we know that $|\mathsf{D}_w|$ is proportional to the size of string w. A machine that transforms w to an encoded D_w, followed by a machine that finds the model of $\mathsf{P} \cup \mathsf{D}_w$ and finally a machine that reads and accepts based on the value of predicate "accept", give us the Turing Machine that decides L in the required running time. □

4.2 Calculating the WFS model of a higher-order Datalog program

In this section, we will describe a Turing Machine with multiple tapes that is created from a fixed Datalog program P and which given any set of input first-order relations D_{in}, calculates the WFS model of $\mathsf{P} \cup \mathsf{D}_{in}$. Furthermore, with the added assumption that the predicates of P are of order up to $k+1$ and the variables of order up to k, we will argue that the running time of said machine is $O(\exp_k(poly(N)))$, where $N = |\mathsf{D}_{in}|$ and with $poly(N)$ we will denote some fixed and controlled by P polynomial of N. For reasons of brevity, we will hide constants under $poly(N)$ and will avoid describing low-level operations such as parsing and decoding the input that obviously don't affect the final run-time complexity.

The general idea and intuition behind the bound in the run-time stems from the following observations.

- We can view predicates as functions of some arity that take as input other functions and when fully applied they output an element from the set $\{\mathit{false}, \mathit{undef}, \mathit{true}\}$. For the purpose of this algorith, we will view predicates in the equivalent way of pairs of monotone-antimonotone, antimonotone-monotone functions. Each function of such pair accepts as input other function pairs and when fully applied it outputs an element from $\{\mathit{false}, \mathit{true}\}$. Therefore each such a function can be represented in memory as a series of entries, one for each possible combination

[1] under some encoding that can be produced in polynomial time w.r.t. $|\mathsf{D}_{in}|$.

of elements that the arguments can take. Each entry ends with a truth value which is the evaluation of the fully applied function. Partial application is handled by creating a new function representation by the old one after fixing some of the initial arguments and collecting a new set of entries.

- For every type ρ of a variable where $order(\rho) = j$, the cardinality of the set $[\![\rho]\!]$ is bounded by $O(\exp_j(poly(N)))$ and it is also$j \leq k$. Therefore it is bounded by $O(\exp_k(poly(N)))$. This bounds the set we draw from to instantiate the variables of a rule and intuitively is exactly what bounds the expressiveness. For example, a first-order predicate that handles arguments of type individual has a representation of polynomial size with respect to N, which is produced by the cartesian product of the sets that correspond to its multiple arguments. Furthermore, the different functions/predicates of such type are exponential (one of three choices for the truth values without considering Fitting-monotonicity restriction). Obviously, this argument is preserved when viewing the meanings of predicates as pairs of functions.

- For each constant predicate p with order up to $k + 1$ we create its meaning or in other words the function (pair of functions) it is assigned to at the current step of the algorithm. Such a function's size can reach up to $O(\exp_k(poly(N)))$ entries again by considering the cardinalities of the sets for the arguments and taking the cartesian product. Notice how we do not need to create all possible such meanings for these types since they never appear as arguments.

- Any polynomial algorithm that acts on $O(\exp_k(poly(N)))$ number of elements will have a final complexity (w.r.t N) of $O(\exp_k(poly(N)))$. We will hide the change of the constants in the resulting polynomial of N using the "$poly()$" notation. This is a straightforward property of the function $\exp_k(x)$.

- The immediate consequence operators can run at most a number of steps proportional to the size that the functions of the constant predicates. That is because the operators produce a monotonically increasing sequence or Fitting-monotonically in the case of $\mathcal{C}_{T_{\mathsf{P}}}$. No change in the representation of all constant predicates after an iteration signals the convergence of the loop. In other words, after each iteration at least one entry at the function representation of at least one predicate must change in an increasing fashion.

We will divide the algorithm into several steps. At the start of the execution assume that the machine has parsed the input D_{in} as a string and has augmented the set of rules of P with the facts in D_{in}. This can obviously be done in polynomial time to $|\mathsf{D}_{in}|$ and is omitted. As stated we will not delve into details about how such a representation is done in the machine since it does not alter the complexity. We divide the algorithm in steps which we describe and argue that each one of them can be performed within $O(\exp_k(poly(N)))$.

4.2.1 Calculate the set of possible elements for each type

We will appoint a separate machine tape for each possible type that an expression that appears in P. This includes every possible sub-expression. Obviously, there is a fixed number of such expressions and it depends only on the fixed program P. We will also not consider expressions of type o since they can't appear as arguments. Finally, we also appoint a separate tape for each constant predicate in P.

For every type ρ and $order(\rho) = j$ and $j \leq k$ that appears in an expression in P, we will calculate and write in its appointed tape the following.

- The set $[\![\rho]\!]^{\mathsf{ma}}$ of monotone-antimonote functions for this type. This is represented in memory as a series of elements each one represented by a function identifier (an increasing integer number) followed by the actual function representation as described below. We will denote the identifier of the function f for the type ρ as $id_\rho(f)$.
- The set $[\![\rho]\!]^{\mathsf{am}}$ of antimonotone-monotone functions for this type. Solely for convenience, we will label the new functions reusing the same counter we used for the $[\![\rho]\!]^{\mathsf{ma}}$ set. This results in identifiers bigger than those corresponding to the previous set.
- The set of tuples of function identifiers $(id_\rho(a), id_\rho(b))$ such that $a, b \in [\![\rho]\!]^{\mathsf{ma}} \cup [\![\rho]\!]^{\mathsf{am}}$ and $a \leq b$, fully describing the partial order of the elements of the set $[\![\rho]\!]^{\mathsf{ma}} \cup [\![\rho]\!]^{\mathsf{am}}$. We will assume that each tuple $(id_\rho(a), id_\rho(b))$ has a flag next to it telling us whether it also holds $a \in [\![\rho]\!]^{\mathsf{ma}}$ and $b \in [\![\rho]\!]^{\mathsf{am}}$, thus making it a valid element of $[\![\rho]\!]^{\mathsf{ma}} \otimes [\![\rho]\!]^{\mathsf{am}}$. It is convenient to store the partial order for the whole $[\![\rho]\!]^{\mathsf{ma}} \cup [\![\rho]\!]^{\mathsf{am}}$ domain as will be seen later.

Given a type $\rho = \rho_1 \to \cdots \to \rho_t \to o$ of arity t we have that $\forall i \in \{1, \cdots, t\}$ $order(\rho_i) = j' < j = order(\rho)$. We will assume that the tape of each ρ_i *is already filled out* which means that we have already calculated the set of elements and their ordering for every sub-type of ρ. This dictates an order of calculation for each type in a recursive fashion defined by the way each type is constructed.

We will also operate under the assumption that the total size of the tape for ρ_i is bounded by $O(\exp_{j'}(poly(N)))$ bits and given the bound in the order for the subtypes, bounded by $\exp_{j-1}(poly(N))$ bits. We show that if this bound (which is controlled by the type's order) holds for each sub-type's tape then it also holds for the tape of the constructed type ρ. It is easy to verify that the case holds for the base type case of ι. Specifically, the number of elements of this type is bounded by the size of the Hebrand Universe which is bounded by $N = |\mathsf{D}_{in}|$ plus a fixed number for the set of constants appearing in P. In other words, the cardinality of the set and of the trivial ordering is proportional to N.

Calculate the elements of $[\![\rho]\!]^{\mathsf{ma}}$

We have that ($\overset{\mathsf{ma}}{\to}$ is right associative)

$$[\![\rho]\!]^{\mathsf{ma}} = [([\![\rho_1]\!]^{\mathsf{ma}} \otimes [\![\rho_1]\!]^{\mathsf{am}}) \overset{\mathsf{ma}}{\to} ([\![\rho_2]\!]^{\mathsf{ma}} \otimes [\![\rho_2]\!]^{\mathsf{am}}) \overset{\mathsf{ma}}{\to} \cdots \overset{\mathsf{ma}}{\to} o]$$

We can ignore the case of ρ_i being the type ι to simplify the analysis since it doesn't affect the following arguments. We can work on a separate tape for each set $[\![\rho_i]\!]^{\mathsf{ma}} \otimes [\![\rho_i]\!]^{\mathsf{am}}$.

For each set $[\![\rho_i]\!]^{\mathsf{ma}} \otimes [\![\rho_i]\!]^{\mathsf{am}}$ we collect each possible element (a_i, a_i') that belongs to it as a tuple of function identifiers $(id_{\rho_i}(a_i), id_{\rho_i}(a_i'))$ from the tape appointed to the type ρ_i. That means we collect every $(id_{\rho_i}(a_i), id_{\rho_i}(a_i'))$ where $a_i \in [\![\rho_i]\!]^{\mathsf{ma}}$ and $a_i' \in [\![\rho_i]\!]^{\mathsf{am}}$ and also $a_i \leq a_i'$. This takes time polynomial to the size of the tape for ρ_i so given the bound in the order of ρ_i it is $O(poly(\exp_{j-1}(poly(N))))$ or equivalently $O(\exp_{j-1}(poly(N)))$.

We proceed by creating every possible function that evaluates to $\{\mathit{false}, \mathit{true}\}$. To create one such function we use the cartesian product of the sets collected above to create every possible instantiation of the arguments.

$$(\llbracket\rho_1\rrbracket^{\mathsf{ma}} \otimes \llbracket\rho_1\rrbracket^{\mathsf{am}}) \times (\llbracket\rho_2\rrbracket^{\mathsf{ma}} \otimes \llbracket\rho_2\rrbracket^{\mathsf{am}}) \times \cdots \times (\llbracket\rho_t\rrbracket^{\mathsf{ma}} \otimes \llbracket\rho_t\rrbracket^{\mathsf{am}})$$

Since we have retrieved all elements that can be slotted as arguments then it is a matter of combining them in every possible way, which can be done within polynomial time w.r.t. the total size of the elements. Assume that the arity for the type ρ is t. Then for each such instantiation such as

$$(id_{\rho_1}(a_1), id_{\rho_1}(a_1')), (id_{\rho_2}(a_2), id_{\rho_2}(a_2')), \cdots, (id_{\rho_t}(a_t), id_{\rho_t}(a_t'))$$

we write down the entry

$$(id_{\rho_1}(a_1), id_{\rho_1}(a_1')), \cdots, (id_{\rho_t}(a_t), id_{\rho_t}(a_t')), T$$

where $T \in \{\mathit{false}, \mathit{true}\}$. The number of possible different instantiations of the arguments is the product of the cardinalities of the sets of elements for each subtype ρ_i therefore bounded by $[\exp_{j-1}(poly(N))]^t$. This bound can be written as $\exp_{j-1}(poly(N))$ by increasing the constants of the polynomial of N. The time needed to write them down for one function is also polynomial to the size of this product. This leads to a running time of

$$O(poly([\exp_{j-1}(poly(N))]^t)) \sim O(\exp_{j-1}(poly(N)))$$

Notice that the size of its function written is also bounded by $O(\exp_{j-1}(poly(N)))$ since any identifier for a subtype has size at most $O(\exp_{j-2}(poly(N)))$ since it is always logarithmic to the number of possible functions for that type.

We proceed with doing this for every possible function. We can use a separate tape to store the set of truth values appointed to the entries as a binary string and thus generate them in increasing order so we do not produce the same function. The number of possible such functions is:

$$2^{|\llbracket\rho_1\rrbracket^{\mathsf{ma}} \otimes \llbracket\rho_1\rrbracket^{\mathsf{am}}| \times |\llbracket\rho_2\rrbracket^{\mathsf{ma}} \otimes \llbracket\rho_2\rrbracket^{\mathsf{am}}| \times \cdots \times |\llbracket\rho_t\rrbracket^{\mathsf{ma}} \otimes \llbracket\rho_t\rrbracket^{\mathsf{am}}|} < 2^{\exp^t_{j-1}(poly(N))} \sim \exp_j(poly(N))$$

Writing all of them down takes time at most

$$O(\exp_j(poly(N))) \times \exp_{j-1}(poly(N)) \sim O(\exp_j(poly(N)))$$

and so is the space we need since each function takes space $O(\exp_{j-1}(poly(N)))$. As we see the current size of the tape we filled so far is totally dominated by the number of possible functions and each function size is irrelevant to the space complexity bound since it is exponentially lower.

Now that we have written every possible function in the form described we need to *filter* those functions that are actually monotone-antimonotone. Remember that we already have the ordering for each sub-type in its tape. For every function written before we perform the following test.
For every entry in the function's representation:

$$(id_{\rho_1}(a_1), id_{\rho_1}(a_1')), \cdots, (id_{\rho_t}(a_t), id_{\rho_t}(a_t')), T$$

we check it against every other entry in it

$$(id_{\rho_1}(b_1), id_{\rho_1}(b'_1)), \cdots, (id_{\rho_t}(b_t), id_{\rho_t}(b'_t)), T'$$

If $\forall i \in \{1, \cdots, t\}$ it is $a_i \leq b_i$, $a'_i \geq b'_i$ and also it is $T' < T$ (*false* < *true*) the test fails and we skip to the next function. If the test passes for all combinations of entries we assign an identifier for the function (increasing positive integer number) since it represents an actual monotone-antimonotone function and we save both the function and the identifier on the tape for the type ρ. If the test fails for at least one pair of entries then we move to the next function. The number of tuple pairs we will compare is bounded by the square of the size of the function so $[\exp_{j-1}(poly(N))]^2 \sim \exp_{j-1}(poly(N))$ and we can collect them in time proportional to the size of the function's representation. For every pair of entries, each point-wise comparison of the elements requires parsing the corresponding tape of the sub-type to check if said pair is included in the partial ordering by searching for it in the appropriate tape. This takes time at most $O(\exp_{j-1}(poly(N)))$ when implemented in the simplest way possible. Thus the total time of testing per function is $O(\exp^2_{j-1}(poly(N))) \times O(\exp_{j-1}(poly(N))) \sim O(\exp_{j-1}(poly(N)))$ and doing it for all $\exp_j(poly(N))$ at most functions gives total running time:

$$\exp_j(poly(N)) \times O(\exp_{j-1}(poly(N))) \sim O(\exp_j(poly(N)))$$

Calculate the elements of $[\![\rho]\!]^{\mathsf{am}}$

We have that

$$[\![\rho]\!]^{\mathsf{am}} = [([\![\rho_1]\!]^{\mathsf{ma}} \otimes [\![\rho_1]\!]^{\mathsf{am}}) \overset{\mathsf{am}}{\to} ([\![\rho_2]\!]^{\mathsf{ma}} \otimes [\![\rho_2]\!]^{\mathsf{am}}) \overset{\mathsf{am}}{\to} \cdots \overset{\mathsf{am}}{\to} o]$$

The process is completely analogous to the one we used to create the set $[\![\rho]\!]^{\mathsf{ma}}$ and so is the running time. The identifier we give each function picks up from where we were left when calculating the $[\![\rho]\!]^{\mathsf{ma}}$ set. We need to change the test to accommodate testing if a function is antimonotone-monotone now.
For every entry in the function's representation:

$$(id_{\rho_1}(a_1), id_{\rho_1}(a'_1)), \cdots, (id_{\rho_t}(a_t), id_{\rho_t}(a'_t)), T$$

check it against every other entry:

$$(id_{\rho_1}(b_1), id_{\rho_1}(b'_1)), \cdots, (id_{\rho_t}(b_t), id_{\rho_t}(b'_t)), T'$$

If $\forall i \in \{1, \cdots, t\}$ it is $a_i \leq b_i$, $a'_i \geq b'_i$ and also it is $T < T'$ the test fails and we skip to the next function.

Create the partial ordering of $[\![\rho]\!]^{\mathsf{ma}} \cup [\![\rho]\!]^{\mathsf{am}}$

The tape for type ρ should at this point already contain each element of $[\![\rho]\!]^{\mathsf{ma}} \cup [\![\rho]\!]^{\mathsf{am}}$. The ordering will be represented as a set of tuples $(id_\rho(f), id_\rho(g))$ where $f \leq g$.

For that we need to compare each pair of functions $f, g \in [\![\rho]\!]^{\mathsf{ma}} \cup [\![\rho]\!]^{\mathsf{am}}$. Given a pair of f, g for every entry in the representation of f

$$(id_{\rho_1}(a_1), id_{\rho_1}(a'_1)), \cdots, (id_{\rho_t}(a_t), id_{\rho_t}(a'_t)), T_{fa}$$

we check it against the corresponding entry (same arguments) in g

$$(id_{\rho_1}(a_1), id_{\rho_1}(a'_1)), \cdots, (id_{\rho_t}(a_t), id_{\rho_t}(a'_t)), T_{ga}$$

If it holds $T_{fa} \leq T_{fb}$ for all entry pairs then we add $(id_\rho(f), id_\rho(g))$ to the end of the tape.

The number of functions we have to compare to each other is bounded by the product of the elements that passed the previous filtering. They are at most $[\text{exp}_j(poly(N))]^2$ pairs and we can collect them at a proportional to their number, running time. Each pair comparison takes time bounded by a polynomial of the size of the functions which is at most $\text{exp}_{j-1}(poly(N))$. This gives us a total running time to create the ordering of

$$O([\text{exp}_j(poly(N))]^2) + [\text{exp}_j(poly(N))]^2 \times O(poly(\text{exp}_{j-1}(poly(N))))$$

which once again is equivalent to

$$O(\text{exp}_j(poly(N)))$$

for a polynomial of N of an appropriately high enough order. Marking the pairs $(id_\rho(f), id_\rho(g))$ in the ordering where it also holds that $f \in [\![\rho]\!]^{\mathsf{ma}}$ and $g \in [\![\rho]\!]^{\mathsf{am}}$ can be done during the previous process with no relevant addition to the running time.

4.2.2 Initialization of the constant predicates

For every constant predicate p with type ρ and order at most $k+1$ on a dedicated tape for the said predicate, we create its initial meaning which is a pair of functions (f_p, g_p) where $f_{\mathsf{p}} \in [\![\rho]\!]^{\mathsf{ma}}$ and $g_{\mathsf{p}} \in [\![\rho]\!]^{\mathsf{am}}$. As per usual the type's ρ_π elements are pairs of functions of the form:

$$[\![\rho_\pi]\!]^{\mathsf{ma}} = [([\![\rho_1]\!]^{\mathsf{ma}} \otimes [\![\rho_1]\!]^{\mathsf{am}}) \overset{\mathsf{ma}}{\to} ([\![\rho_2]\!]^{\mathsf{ma}} \otimes [\![\rho_2]\!]^{\mathsf{am}}) \overset{\mathsf{ma}}{\to} \cdots \overset{\mathsf{ma}}{\to} o]$$

and:

$$[\![\rho_\pi]\!]^{\mathsf{am}} = [([\![\rho_1]\!]^{\mathsf{ma}} \otimes [\![\rho_1]\!]^{\mathsf{am}}) \overset{\mathsf{am}}{\to} ([\![\rho_2]\!]^{\mathsf{ma}} \otimes [\![\rho_2]\!]^{\mathsf{am}}) \overset{\mathsf{am}}{\to} \cdots \overset{\mathsf{am}}{\to} o]$$

We repeat the previous process for the type ρ_p with a crucial difference. We do not create every possible function by allowing the truth values in the entries to be of of any choice. We only create exactly two elements.

- We set f_{p} to be the bottom element for the type which is a monotone-antimonotone function where each entry ends with *false* in its representation.
- We set g_{p} to be the top element which is an antimonotone-monotone function where each entry ends with *true* in its representation.

With a similar analysis as before and given that the size of the functions created for any constant predicate of order at most $k+1$ is at most $O(\text{exp}_k(poly(N)))$, we get a total running time for all predicates at $O(\text{exp}_k(poly(N)))$. There is a fixed number of constant predicates [2] and we create two elements for each one of them. Notice that if we were to instantiate every possible element of a type of order $k+1$ we would require time of $O(\text{exp}_{k+1}(poly(N)))$.

[2] In the case where we allow new predicate definitions in the input database D_{in} we can still argue that there are polynomially many of them w.r.t the total combined size of the program and the database.

4.2.3 Perform the bottom-up iterative procedure

We have for the stable revision operator that

$$\mathcal{C}_{T_{\mathsf{P}}}(I,J) = (J^{\downarrow}, I^{\uparrow}) = (\mathit{lfp}([T_{\mathsf{P}}(\cdot, J)]_1), \mathit{lfp}([T_{\mathsf{P}}(I, \cdot)]_2))$$

and we get the sequence

$$\begin{array}{lcl} (I_0, J_0) & = & (\bot, \top) \\ (I_{\lambda+1}, J_{\lambda+1}) & = & \mathcal{C}_{T_{\mathsf{P}}}(I_\lambda, J_\lambda) \end{array}$$

For our functionless language, we can assume that there is a point in the sequence where $(I_n, J_n) = \mathcal{C}_{T_{\mathsf{P}}}(I_n, J_n)$. We will describe the process of applying $\mathcal{C}_{T_{\mathsf{P}}}$ to a (I_n, J_n) and argue about the running time of each application step.

a) Calculate $I_{n+1} = \mathit{lfp}([T_{\mathsf{P}}(\cdot, J_n)]_1)$:

Assume that we have already (I_n, J_n) and we need to calculate I_{n+1}. We have to calculate the following sequence.

$$\begin{array}{lcl} I^1_{n+1} & = & [T_{\mathsf{P}}(\bot, J_n)]_1 \\ I^2_{n+1} & = & [T_{\mathsf{P}}(I^1_{n+1}, J_n)]_1 \\ I^3_{n+1} & = & [T_{\mathsf{P}}(I^2_{n+1}, J_n)]_1 \\ & \cdots & \end{array}$$

For each constant predicate p, we create a copy of its current meaning $(I_n(\mathsf{p}), J_n(\mathsf{p}))$ for safekeeping before we proceed. Remember that after we produce I_{n+1} we still need I_n to create J_{n+1}. We also create another copy of every $I_n(\mathsf{p})$ which we initialize to the bottom element by setting the truth values of the entries to *false*. In other words, we create $I^0_{n+1}(\mathsf{p})$ before we initiate the main loop.

We then describe the general iteration step and its complexity:

$$I^{k+1}_{n+1} \quad = \quad [T_{\mathsf{P}}(I^k_{n+1}, J_n)]_1$$

For the immediate two-valued operator T_{P} we have:

$$T_{\mathsf{p}}(I,J)(\mathsf{p})d_1 \cdots d_t =$$

$$\bigvee\{\bigwedge [\![\mathsf{L}_i]\!]_{s[\mathsf{V}_1/d_1,\ldots,\mathsf{V}_n/d_t]}(I,J) \mid (\mathsf{p}\,\mathsf{V}_1 \cdots \mathsf{V}_t \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m) \in \mathsf{P},\, s \text{ a pair Herbrand state}\}$$

We create the following loop to implement this single step of applying the operator.
For each constant p make a copy of $I^k_{n+1}(\mathsf{p})$ to alter.
For each constant predicate p:
For every rule in P with p on the head such as:

$$\mathsf{p}\,\mathsf{V}_1 \cdots \mathsf{V}_t \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$$

and

$$\mathsf{L}_i = \begin{cases} \mathsf{E}_i \\ {\sim}\mathsf{E}_i \end{cases}$$

with $\{\mathsf{E}_1, \cdots, \mathsf{E}_m\}$ atoms and $V_1, \cdots, V_{t'}$, $t' \geq t$ being the set of the different variables appearing in the rule (formal or existential) with corresponding types $\rho_1, \cdots, \rho_{t'}$:
For every possible instantiation for the set of variables so as each variable V_i is assigned to a tuple of function identifiers $(id_{\rho_i}(v_i), id_{\rho_i}(v'_i))$ where $(v_i, v'_i) \in [\![\rho_i]\!]^{\mathsf{ma}} \otimes [\![\rho_i]\!]^{\mathsf{am}}$ do:

- For every constant predicate q in the body of the rule with $order(\mathsf{q}) \leq k$ match the current meaning of q against the list of functions in the tape of ρ_{q} to find a matching pair of identifiers of an element in $[\![\rho_{\mathsf{q}}]\!]$. If $order(\mathsf{q}) = k + 1$ assign a fixed set of special identifiers which signal that we will find the corresponding pair of functions in the tape of q (remember we didn't create all instantiations for types with order $k+1$). This is done so we can treat constant predicates appearing in argument positions in a unifying way with what we do for the instantiated variables.

- For every E_i calculate its value under the current instantiation of the variables. We allow partial application so the syntactical tree of E_i can have arbitrary height but that is purely program dependent. In general, we can calculate the value of an expressions **recursively** as follows:

  – For a fully applied expression of the form "$\mathsf{e}\ a_1 \dots a_t$" assume each argument expression a_i is already calculated which means we have a pair of function identifiers for every a_i as (l_i, r_i). We also have a pair of identifiers (f, g) for e. We use f to locate the monotone-antimonotone function in the appropriate tape and after reaching its function representation we use the pairs-identifiers of the arguments to locate the corresponding entry and retrieve the truth value $(\mathit{false}, \mathit{true})$. This essentially calculates fully applied $f(l_1, r_1) \dots (l_t, r_t)$. We do the same for f_2 and then we combine the two results to get

  $$(f\ (l_1, r_1) \dots (l_t, r_t), g\ (l_1, r_1) \dots (l_t, r_t))$$

  We can then convert the result to the 3-valued domain using the function τ for type o.

  $\tau_o^{-1}(\mathit{false},\mathit{false}) = \mathit{false}, \tau_o^{-1}(\mathit{true}, \mathit{true}) = \mathit{true}, \tau_o^{-1}(\mathit{false}, \mathit{true}) = \mathit{undef}$

  – For a partially applied expression "$\mathsf{e}\ a_1 \dots a_{t'}$" with $t' < t = arity(\mathsf{e})$ instead of a pair of truth values we have to return a pair of function identifiers. We again assume each expression a_i is already calculated (recursive approach) so we have a pair of function identifiers (l_i, r_i) for every a_i and also a pair of identifiers (f, g) for e.
  We use f to locate the monotone-antimonotone function in the appropriate tape. Then using the identifiers for the first t' arguments we collect a set of all the entries in f that start with those elements. Deleting the fixed first t' from each entry in the set gives us a new function representation. We use the type of $\mathsf{e}\ a_1 \dots a_{t'}$ to parse the appropriate tape and match the new function representation we found to an identifier f'. Since $\mathsf{e}\ a_1 \dots a_{t'}$ is an expression of the program we know that we have created all elements of this type in the earlier stages. We then use g to locate the antimonotone-monotone function in the appropriate tape. By a similar process, we get the second function identifier g'.
  We return the identifiers pair (f', g').

  – For an expression of the form $(\mathsf{E}_1 \approx \mathsf{E}_2)$ since our language lack functions, each one of $\mathsf{E}_1, \mathsf{E}_2$ is either a variable or a constant of type ι. The meaning of the expression under current instantiation is the same regardless of interpretations and can be calculated easily. We can then convert the result again to the 3-valued domain using the function τ for type o.

- Calculate every L_i under this instantiation by using E_i and the inverse rules *true*$^{-1}$ = *false*, *false*$^{-1}$ = *true* and *undef*$^{-1}$ = *undef*. We choose to convert the final result in the 3-valued truth domain for convenience but it is not necessary.

- Calculate and return $T_{new} = [\tau_o(\bigwedge \mathsf{L}_i)]_1$ by taking the minimum truth value (*false* $<$ *undef* $<$ *true*) from those calculated in the previous step for each L_i, transforming it with τ function and projecting the first element.

- Go at the copy of $I'_i(\mathsf{p})$ and locate the entry

$$(id_{\rho_1}(v_1), id_{\rho_1}(v'_1)), \cdots, (id_{\rho_{\mathsf{V}_t}}(v_t), id_{\rho_{\mathsf{V}_t}}(v'_t)), T_{old}$$

  which match for the current instantiation (that is $s(V_i) = (id_{\rho_i}(v_i), id_{\rho_i}(v'_i))$). If $T_{new} > T_{old}$ then replace the truth value with T_{new}. This implements the $\bigvee$ over all rules as we consider them one by one.

After this nested loop is done, for every constant predicate p set $I^{k+1}_{n+1}(\mathsf{p})$ to be the copy of $I^k_{n+1}(\mathsf{p})$ we have been altering.

For the running time of one step, we can argue about the following. The number of constant predicates is fixed and depends on P. The number of rules is $O(N)$ if we take into account the variable number of facts that appear in D_{in}. The number of possible combinations we have for an instantiation of any of the variables $\mathsf{V}_1, \cdots, \mathsf{V}_t$ is bounded by $\exp_k(poly(N))$ and the time to collect them is polynomial to that, therefore

$$O(poly([\exp_k(poly(N))]^t))$$

The time it takes to calculate an expression like $\mathsf{e}\ a_1 \ldots a_{t'}$ given the size of the functions we manipulate and the fact we implement polynomial algorithms in regard to their size, is also bounded by $O(\exp_k(poly(N)))$. There is also a fixed number of such expressions we need to calculate to find each atom E_i and therefore a fixed total number of them per rule. Finally the act of updating a truth value in a copy of $I^k_{n+1}(\mathsf{p})$ takes also time bounded by $O(\exp_k(poly(N)))$. Putting all this together, the total running time of the loop is

$$O(poly([\exp_k(poly(N))]^m)) + O(N) \times O([\exp_k(poly(N))]^m) \times (O(\exp_k(poly(N)))$$

$$\sim O(\exp_k(poly(N))$$

for a large enough fixed polynomial of N.

For the total running time of finding $I_{n+1} = lfp([T_\mathsf{P}(\cdot, J_n)]_1)$ we need to argue about how many steps we will need to take till convergence. Each step as shown above takes $O(\exp_k(poly(N))$ time. The sequence $I^0_{n+1}, I^1_{n+1}, I^2_{n+1}, \cdots$ is increasing. That means that each time we perform a step for at least one constant predicate p it must hold that $I^{k+1}_{n+1}(\mathsf{p}) > I^k_{n+1}(\mathsf{p})$ or equivalently at least one entry in the representation of the current altered $I^{k+1}_{n+1}(\mathsf{p})$ has its truth value increased compared to the same entry in $I^k_{n+1}(\mathsf{p})$. Taking into account that the number of constant predicates is fixed, let's say C, and that the bound of their function representation is $\exp_k(poly(N))$ entries we get that at $C \times \exp_k(poly(N)) \sim O(\exp_k(poly(N)))$ steps every one of them must have stopped increasing. So at most $O(\exp_k(poly(N)))$ steps we have converged. Thus the total running time of finding $I_{n+1} = \mathit{lfp}([T_{\mathsf{P}}(\cdot, J_n)]_1)$ is the number of steps $\times$ running time of one step.

$$O(\exp_k(poly(N)) \times O(\exp_k(poly(N)) \sim O(\exp_k(poly(N))$$

Once again for a polynomial of N of high enough order, the time bound for the whole process is $O(\exp_k(poly(N))$ which means we are still within $k - \mathsf{EXPTIME}$ class.

b) Calculate $J_{n+1} = \boldsymbol{lfp}([T_{\mathsf{P}}(I_n, \cdot)]_2)$:

The fixed point is now given by the following sequence

$$\begin{array}{rcl} J^1_{n+1} & = & [T_{\mathsf{P}}(I, I^*_n)]_2 \\ J^2_{n+1} & = & [T_{\mathsf{P}}(I, J^1_{n+1}]_2 \\ & \cdots & \\ J^{\alpha+1}_{n+1} & = & [T_{\mathsf{P}}(I, J^{\alpha}_{n+1})]_2 \\ & \cdots & \end{array}$$

The procedure of finding it out is the same as described for finding $I_{n+1} = \mathit{lfp}([T_{\mathsf{P}}(\cdot, J_n)]_1)$ and gives the same running time. For this reason, it will be omitted. There is a difference in the initialization that we have to consider. In the sequence, we take as a starting point

$$J^0_{n+1} = I^*_n$$

where I^*_n is the least interpretation belonging in $\mathcal{H}^{\mathsf{am}}_{\mathsf{P}}$, such that $I^*_n \geq I_n$. For every constant predicate p we need to calculate $I^*(\mathsf{p})$ as the least antimonotone-monotone function such that $I^*(\mathsf{p}) > I_n(\mathsf{p})$. This can be done in $O(\exp_k(poly(N))$ by simply applying the following iterative procedure. For every constant predicate p of arity t.

- Set initially $I^*_n(p) = I_n(p)$
- For every pair of entries in $I^*_n(p)$

$$(id_{\rho_1}(a_1), id_{\rho_1}(a'_1)), \cdots, (id_{\rho_t}(a_t), id_{\rho_t}(a'_t)), T_1$$

$$(id_{\rho_1}(b_1), id_{\rho_1}(b'_1)), \cdots, (id_{\rho_t}(b_t), id_{\rho_t}(b'_t)), T_2$$

  If $\forall i$ holds $a_i \leq b_i$ and $a'_i \geq b'_i$ and also $T_1 < T_2$ then change T_1 to T_2 in the first entry.

With similar arguments and using the space-bound in the representation of the functions, we get a running time of $O(\exp_k(poly(N))$ for this procedure. The calculation of the fixed point is then done in the same way as before.

Final complexity of bottom-up procedure

We have shown that the step

$$(I_{n+1}, J_{n+1}) = \mathcal{C}_{T_{\mathsf{P}}}(I_n, J_n)$$

takes time $O(\exp_k(poly(N))$ for a fixed polynomial of N. We need to argue about how many iterations of performing this "outer" step we will need to apply till the interpretations pair converges to a fix-point. The bound follows from the fact that $\mathcal{C}_{T_{\mathsf{P}}}$ is Fitting-monotonic for the pairs (I_n, J_n) we consider. It specifically holds that:

$$(I_n, J_n) \preceq (I_{n+1}, J_{n+1})$$

That means after each step for at least one constant predicate p it is $(I_n, J_n)(p) \preceq (I_{n+1}, J_{n+1})(p)$. By the bound $O(\exp_k(poly(N))$ in the representation of each constant predicate and the fact that the number of them is a fixed number we have that we will apply $\mathcal{C}_{T_\mathsf{P}}$ at most $\exp_k(poly(N))$ times. Taking into account the complexity of performing one step we have a total running time:

$$\exp_k(poly(N)) \times O(\exp_k(poly(N))) \sim O(\exp_k(poly(N)))$$

Once again for a polynomial of N of high enough order, we are still within $k-\mathsf{EXPTIME}$ class for the whole calculation.

CHAPTER 5
NEGATIVE DATALOG WITHOUT PARTIAL APPLICATION

In this chapter, we prove the second row of the table in 1. This row contains the results for the language fragments $\mathcal{H}_k^{\neg,\exists}$ for $k = 1, 2, \ldots$.

partial application	negation	h.o. existential variables	order 1	order 2	order 3	$\cdots$	order ∞
NO	YES	YES	PTIME	EXPTIME	$2-$EXPTIME	$\cdots$	ELEM.

5.1 Theorems

The main theorem is the following.

Theorem 5.1. *For every $k \geq 1$ $\mathcal{H}_k^{\neg,\exists}$ captures exactly $(k-1)-$* EXPTIME.

Proof. Lemma 5.2 shows that for every $k \geq 1$ $\mathcal{H}_k^{\neg,\exists}$ expresses at least $(k-1)-$ EXPTIME. Corollary 5.4 that is derived from Lemma 5.3 proves that $\mathcal{H}_k^{\neg,\exists}$ can express at most $(k-1)-$ EXPTIME. □

We present now the two necessary lemmas that we need to prove the main theorem.

Lemma 5.2. *Let there be a deterministic Turing Machine that decides a language L in $(k-1)-$* EXPTIME*, $k \geq 1$. Then, there exists a program* P *that belongs in $\mathcal{H}_k^{\neg,\exists}$ that decides L.*

Proof. For $k = 1$ this has been shown numerous times since even standard Datalog expresses PTIME on ordered Databases. Also, notice that the Well-Founded model of semantics for higher-order Datalog presented here reduces to the classical Well- Founded semantics in the literature for first-order Datalog which is also shown in the literature to fully coincide with positive semantics in the absence of negation in the program.

For $k \geq 2$ the Lemma is proven by a simulation of a general Turing Machine that decides a language L in $(k-1)-$ EXPTIME through a Datalog program that belongs in $\mathcal{H}_k^{\neg,\exists}$. The simulation is given in the following section. □

Lemma 5.3. *Let* P *be a Datalog program in* $\mathcal{H}_k^{\neg,\exists}$ *with* $k \geq 2$. *Then there exists a Turing machine that for any set of input (first-order) relations* D_{in}*, given* D_{in} *as input, it can calculate the WFS model of* $\mathsf{P} \cup \mathsf{D}_{in}$ *within time* $O(\exp_{k-1}(n^q))$*, where* n *is the size of* D_{in} *and* q *is a constant that depends only on* P.

We can show it by reusing the more general (includes partial application) Turing machine described in section 4.2 since it has the running time complexity that the lemma requires. This lemma gives us the following corollary result.

Corollary 5.4. *Let* P *in* $\mathcal{H}_k^{\neg,\exists}$ *be a program that decides* L. *There exists a Turing machine that decides* L *and runs in time* $O(\exp_{k-1}(n^q))$*, where* n *is the length of the input string and* q *is a constant that depends only on* P.

Let P in $\mathcal{H}_k^{\neg,\exists}$ be a program that decides L. There exists a Turing machine that for every string w and corresponding input relation D_w finds the model of $\mathsf{P} \cup \mathsf{D}_w$. Furthermore, the machine does that in time $O(\exp_{k-1}(n^q))$ where $n = |\mathsf{D}_w|$ and by the imposed form of D_w we know that $|\mathsf{D}_w|$ is proportional to the size of string w. A machine that transforms w to D_w (encoded), followed by a machine that finds the model of $\mathsf{P} \cup \mathsf{D}_w$ and finally a machine that reads and accepts based on the value of predicate "accept", give us the Turing Machine that decides L in the required running time.

In the next section, we provide a simulation of a Turing machine with programs in $\mathcal{H}^{\neg,\exists}$.

5.2 Simulation with $\mathcal{H}^{\neg,\exists}$ Datalog

Assume a Turing machine M that decides L in $O(\exp_k n^q)$ (n the size of the input string). Then there exists an integer d such that for every input string w with $n = |w|$, M terminates at most $\exp_k n^d$ steps. The datalog program we will present gives the correct result in its "accept" predicate by simulating $\exp_k n^d$ steps of M. For that, we first need to find a way to simulate counting from 0 to $\exp_k n^d$. This will be done in the following steps.

5.2.1 Counting to n^d

We define the predicate `base_zero` which is true for the constant 0 which by our convention is always the starting number and the first argument of the first in order tuple in the `input` relation. Then we define `base_last` which is true for the number $n-1$ which is the first argument of the last in order tuple in the input relation. Then we have `base_succ` which takes two arguments of type ι and succeeds only if the second argument is the successor of the first in the ordering. From `base_succ` we can define `base_pred` which succeeds if its second argument is the predecessor of its first argument:

```
base_zero 0.
base_last I     ←  (input I X end).
base_succ I J  ←  (input I X J),(input J,A,K).
base_pred I J  ←  (base_succ J I).
```

Given the above predicates, we can simulate counting from 0 up to $n-1$. We can extend the range of the numbers we can support up to $n^d - 1$. This will be done by introducing predicates that take d distinct arguments. We will view for convenience these arguments as d-tuples. We use the notation $\overline{\mathtt{X}}$ to represent the sequence of d arguments $\mathtt{X}_1, \ldots, \mathtt{X}_d$. The number representation comes naturally in the sense that tuple $\overline{\mathtt{X}} = \mathtt{X}_1, \ldots, \mathtt{X}_d$ represents number $\mathtt{X}_d + \mathtt{X}_{d-1} \times n + \cdots + \mathtt{X}_1 \times n^{d-1}$.

We define the predicate `tuple_zero` that succeeds for 0 in this new tuple representation, `tuple_last` which succeeds for the tuple representing $n^d - 1$ and `tuple_base_last` that succeeds on the tuple that represents $n - 1$. These definitions use the previously defined predicates.

```
tuple_zero $\overline{X}$          $\leftarrow$  (base_zero $X_1$),...,(base_zero $X_d$).
tuple_last $\overline{X}$          $\leftarrow$  (base_last $X_1$),...,(base_last $X_d$).
tuple_base_last $\overline{X}$     $\leftarrow$  (base_zero $X_1$),...,(base_zero $X_{d-1}$),
                          (base_last $X_d$).
```

Then we have to define the new successor predicate for the new representation. This `tuple_succ` predicate takes as arguments two tuples and succeeds if the second is the successor of the first. We will use the equality over individual types to define it and we will need d rules that have as arguments two tuples with d elements each:

```
tuple_succ $\overline{X}$ $\overline{Y}$   $\leftarrow$   ($X_1$ $\approx$ $Y_1$),...,($X_{d-1}$ $\approx$ $Y_{d-1}$),
                   (base_succ $X_d$ $Y_d$).
tuple_succ $\overline{X}$ $\overline{Y}$   $\leftarrow$   ($X_1$ $\approx$ $Y_1$),...,($X_{d-2}$ $\approx$ $Y_{d-2}$),
                   (base_succ $X_{d-1}$ $Y_{d-1}$),
                   (base_last $X_d$),
                   (base_zero $Y_d$).
              ...
tuple_succ $\overline{X}$ $\overline{Y}$   $\leftarrow$   (base_succ $X_1$ $Y_1$),
                   (base_last $X_2$),...,(base_last $X_d$),
                   (base_zero $Y_2$),...,(base_zero $Y_d$).
```

Then `tuple_pred` can be defined as follows:

```
tuple_pred $\overline{X}$ $\overline{Y}$   $\leftarrow$   tuple_succ $\overline{Y}$ $\overline{X}$.
```

It is convenient we define some auxiliary predicates to reduce code repetition. The `less_than` predicate is defined as follows:

```
less_than $\overline{X}$ $\overline{Y}$   $\leftarrow$   (tuple_succ $\overline{X}$ $\overline{Y}$).
less_than $\overline{X}$ $\overline{Y}$   $\leftarrow$   (tuple_succ $\overline{X}$ $\overline{Z}$),(less_than $\overline{Z}$ $\overline{Y}$).
```

The predicate `tuple_non_zero` that succeeds if its argument is not equal to zero:

```
tuple_non_zero $\overline{X}$   $\leftarrow$   (tuple_zero $\overline{Z}$),(less_than $\overline{Z}$ $\overline{X}$).
```

We will also require another helper predicate that will distinguish which $\overline{\mathtt{X}}$ are valid representations of numbers. This is an artificial issue that only arises from the fact that the universe besides $\mathtt{0}, \ldots, \mathtt{N} - \mathtt{1}$ also contains the constants "a", "b" and "end" which do not belong in the ordering.

```
valid_tuple $\overline{X}$   $\leftarrow$   (tuple_zero $\overline{X}$).
valid_tuple $\overline{X}$   $\leftarrow$   (tuple_pred $\overline{X}$ $\overline{Y}$),(valid_tuple $\overline{Y}$).
```

These predicates are enough to get us up to the natural number $n^d - 1$. We now move to the next step of defining our counting scheme.

5.2.2 Counting to 2^{n^d}

To go even higher we will need second-order predicates which handle first-order predicates as "numbers". Like in [1] we use the idea drawn from [7] that represents numbers by a function $f : \{0, \dots, n^d - 1\} \to \{0, 1\}$. The function can be seen as a binary string and thus it is straightforward to see which is the represented number and the range that can be represented this way (we assume that $f(0)$ is the least significant bit of the number).

To do this with Datalog we will use a first-order predicate that takes as an argument a tuple as described previously. In contrast to [1] we can use negation to our advantage. Let's say a number n is represented by the predicate p_n . We want that (p_n $\overline{\texttt{X}}$) succeeds iff in the binary representation of n the bit at position indicated by the number that $\overline{\texttt{X}}$ represents, is equal to 1. Similarly $\sim$ (p_n $\overline{\texttt{X}}$) succeeds iff the bit at position indicated by the number that $\overline{\texttt{X}}$ represents is equal to 0.

We will now proceed with defining the relevant predicates we are going to use. We start with last$_1$ which represents the last number which is $2^{n^d} - 1$.

$$\texttt{last}_1\ \overline{\texttt{X}} \quad \leftarrow \quad \texttt{(valid\_tuple}\ \overline{\texttt{X}}\texttt{).}$$

The predicate last$_1$ must succeed for every argument that represents a valid zero-order number. To discard second-order functions that do not represent numbers we use we use the following constant predicates.

$$\begin{array}{lcl} \texttt{non\_number}_1\ \texttt{N} & \leftarrow & \texttt{(N}\ \overline{\texttt{X}}\texttt{)},\sim\texttt{(valid\_tuple}\ \overline{\texttt{X}}\texttt{).} \\ \texttt{is\_number}_1\ \texttt{N} & \leftarrow & \sim\texttt{(non\_number}_1\ \texttt{N).} \end{array}$$

Notice however that the predicate is_number$_1$ in the three-valued WFS model should also succeed for predicates such as ones that do not "contain" non-number zero order tuples (output *false* with them as arguments) but output $undef$ when given as argument a valid tuple. This can happen since we will use variable instantiation and higher-order existential variables but is not a problem since those instantiations should fail to make the predicates for the predecessor and successor succeed.

We now define the second-order predicate is_zero$_1$. This essentially checks if its argument which is a first-order predicate is the one that represents the number 0. The predicate non_zero$_1$ accordingly succeeds if its argument does not represent the number zero.

$$\texttt{is\_zero}_1\ \texttt{N} \quad \leftarrow \quad \texttt{(is\_number}_1\ \texttt{N)},\sim\texttt{(non\_zero}_1\ \texttt{N).}$$

$$\texttt{non\_zero}_1\ \texttt{N} \quad \leftarrow \quad \texttt{(is\_number}_1\ \texttt{N),(valid\_tuple}\ \overline{\texttt{X}}\texttt{),(N}\ \overline{\texttt{X}}\texttt{).}$$

Now we define the predicate is_last$_1$ that will succeed if its argument is the relation that represents the last number. The predicate non_last$_1$ tells us the opposite, that its argument does not represent the last number.

$$\texttt{is\_last}_1\ \texttt{N} \quad \leftarrow \quad \texttt{(is\_number}_1\ \texttt{N)},\sim\texttt{(non\_last}_1\ \texttt{N).}$$

$$\texttt{non\_last}_1\ \texttt{N} \quad \leftarrow \quad \texttt{(is\_number}_1\ \texttt{N),(valid\_tuple}\ \overline{\texttt{X}}\texttt{)},\sim\texttt{(N}\ \overline{\texttt{X}}\texttt{).}$$

Finally, we need the auxiliary predicate (exists_one_to_right$_1$ N $\overline{\texttt{X}}$) which we will use later. It succeeds if there is a bit equal to 1 in the binary representation of the represented number, in the current or a following position, that $\overline{\texttt{X}}$ represents.

```
    exists_one_to_right$_1$ N $\overline{X}$  ←  (N $\overline{X}$).
    exists_one_to_right$_1$ N $\overline{X}$  ←  (tuple_pred $\overline{X}$ $\overline{Y}$),
                                    (exists_one_to_right$_1$ N $\overline{Y}$).
```

Now we have what we need to define the predecessor of a number. The case we study is very similar to the first-order case. Intuitively for the number `M` to be the predecessor of `N`, starting from the left of the binary representation, `M` has to be equal in bit value with `N` in the current position if in `N` there is at least one "1" to the right. Otherwise, it needs to have the inverse bit value.

```
pred$_1$ N M            ←  (is_number$_1$ M),(non_zero$_1$ N),(tuple_last $\overline{X}$),
                         (hpred$_1$ N M $\overline{X}$).

hpred$_1$ N M $\overline{X}$          ←  (tuple_zero $\overline{X}$),(bit_unequal$_1$ N M $\overline{X}$).
hpred$_1$ N M $\overline{X}$          ←  (tuple_pred $\overline{X}$ $\overline{Y}$),(exists_one_to_right$_1$ N $\overline{Y}$),
                         (bit_equal$_1$ N M $\overline{X}$), (hpred$_1$ N M $\overline{Y}$).
hpred$_1$ N M $\overline{X}$          ←  (tuple_pred $\overline{X}$ $\overline{Y}$),∼(exists_one_to_right$_1$ N $\overline{Y}$),
                         (bit_unequal$_1$ N M $\overline{X}$), (hpred$_1$ N M $\overline{Y}$).

bit_equal$_1$ N M $\overline{X}$      ←  (N $\overline{X}$),(M $\overline{X}$).
bit_equal$_1$ N M $\overline{X}$      ←  ∼(N $\overline{X}$),∼(M $\overline{X}$).

bit_unequal$_1$ N M $\overline{X}$    ←  (N $\overline{X}$),∼(M $\overline{X}$).
bit_unequal$_1$ N M $\overline{X}$    ←  ∼(N $\overline{X}$),(M $\overline{X}$).
```

We can define `succ`$_1$ which gives the successor of a given number like this:

```
    succ$_1$ N M  ←  (is_number$_1$ M),(non_last$_1$ N), (pred$_1$ M N).
```

We need to test for equality of two numbers `N` and `M`. Intuitively we compare them bit by bit, starting from the leftmost possible position and moving to the right.

```
  equal$_1$ N M          ←  (tuple_last $\overline{X}$),(equal_test$_1$ N M $\overline{X}$).

  equal_test$_1$ N M $\overline{X}$  ←  (tuple_zero $\overline{X}$),(bit_equal$_1$ N M $\overline{X}$).
  equal_test$_1$ N M $\overline{X}$  ←  (bit_equal$_1$ N M $\overline{X}$),(tuple_pred $\overline{X}$ $\overline{Y}$),
                          (equal_test$_1$ N M $\overline{Y}$).
```

Finally, we will need the "less-than" relation, defined as follows:

```
less_than$_1$ N M  ←  (is_zero$_1$ N),(non_zero$_1$ M).
less_than$_1$ N M  ←  (non_zero$_1$ N),(non_zero$_1$ M),
                   (pred$_1$ N N'), (pred$_1$ M M'), (less_than$_1$ N' M').
```

Notice how the order of predicates like pred$_1$ is 2. That is because they manipulate first-order predicates. As a rule of thumb, the number plus one in the index of the predicate name will signal their order. The predicate `last`$_1$ is an exception since the index is equal to its order. Finally, this concludes the counting module for the second-order case. We can now present the framework for arbitrary order.

5.2.3 Counting to $exp_{k+1}n^d$, $k > 1$

We assume that all predicate definitions relevant to counting numbers up to $\exp_k n^d$ already have been added to the program. We build on top of them to reach $exp_{k+1}n^d$.

We start again with the predicate representing the last number.

```
last_{k+1} X   ←   (is_number_k X).
```

To test if a $(k+1)$-order relation represents a number we use we use the following constant predicates.

```
non_number_{k+1} N   ←   (N X),(non_number_k X).
non_number_{k+1} N   ←   (equal_k X Y),(N X),∼(N Y).
non_number_{k+1} N   ←   (equal_k X Y),∼(N X),(N Y).
is_number_{k+1} N    ←   ∼(non_number_1 N).
```

We now define the predicate `is_zero`$_{k+1}$. This again checks if its argument, which is now a higher-order predicate is the one that represents the number 0.

```
is_zero_{k+1} N    ←   (is_number_{k+1} N),∼(non_zero_{k+1} N).
non_zero_{k+1} N   ←   (is_number_1 N),(is_number_k X),(N X).
```

We define the predicates that check if the argument is the last number.

```
is_last_{k+1} N    ←   (is_number_{k+1} N),∼(non_last_{k+1} N).
non_last_{k+1} N   ←   (is_number_1 N,(is_number_k X),∼(N X).
```

Then we need `exists_one_to_right`$_{k+1}$ which intuitively behaves like the one we demonstrated in the second-order case. It succeeds if there is a bit equal to 1 in the binary representation of the represented number, in a position at and after the number that X represents.

```
exists_one_to_right_{k+1} N X   ←   (N X).
exists_one_to_right_{k+1} N X   ←   (pred_k X Y),
                                    (exists_one_to_right_{k+1} N Y).
```

We proceed with the predecessor predicate for this order.

```
pred_{k+1} N M             ←   (is_number_{k+1} M),(non_zero_{k+1} N),
                               (hpred_{k+1} N M last_k).

hpred_{k+1} N M X          ←   (is_zero_k X),(bit_unequal_{k+1} N M X).
hpred_{k+1} N M X          ←   (pred_k X Y),(exists_one_to_right_{k+1} N Y),
                               (bit_equal_{k+1} N M X),(hpred_{k+1} N M Y).
hpred_{k+1} N M X          ←   (pred_k X Y),∼(exists_one_to_right_{k+1} N Y),
                               (bit_unequal_{k+1} N M X), (hpred_{k+1} N M Y).

bit_equal_{k+1} N M X      ←   (N X),(M X).
bit_equal_{k+1} N M X      ←   ∼(N X),∼(M X).

bit_unequal_{k+1} N M X    ←   (N X),∼(M X).
bit_unequal_{k+1} N M X    ←   ∼(N X),(M X).
```

We can define the successor of a given number like this:

```
succ_{k+1} N M  ←  (is_number_{k+1} M),(non_last_{k+1} N), (pred_{k+1} M N).
```

To test for equality we need:

```
equal_{k+1} N M          ←  (is_number_{k+1} N),(is_number_{k+1} M),
                             (equal_test_{k+1} N M last_k).

equal_test_{k+1} N M X   ←  (is_zero_{k+1} X),(bit_equal_{k+1} N M X).
equal_test_{k+1} N M X   ←  (pred_k X Y),(bit_equal_{k+1} N M X),
                             (equal_test_{k+1} N M Y).
```

Finally, we show the "less-than" relation:

```
less_than_{k+1} N M  ←  (is_zero_{k+1} N),(non_zero_{k+1} M).
less_than_{k+1} N M  ←  (non_zero_{k+1} N),(non_zero_{k+1} M),
                         (pred_{k+1} N N'), (pred_{k+1} M M'),
                         (less_than_{k+1} N' M').
```

This concludes the counting module where numbers are represented by higher-order predicates. For any finite k we can build a finite program like shown that can simulate the "counting" up to $exp_k n^d$.

5.2.4 Simulating the Turing Machine

In this session, we will use this framework developed previously to do the actual simulation of the Turing machine. Specifically, we can use an $k+1$-order Datalog program to simulate a k-exponential-time-bounded Turing machine. The program is similar to the one given in [1] with the notable exception that now any instance of partial application is removed and instead, existential higher-order variables are used. Both negation and higher-order existential variables are needed to be able to fully remove partial application from the simulation program of such a Turing machine unless $k = 0$.

Firstly, we require a predicate that transforms the numbers that appear in the input relation and are of type ι, to their k-order relation counterpart. The predicate `base_to_higher`$_k$ `M X` succeeds if the second argument is a predicate that represents the number `M`. When $k = 2$ we have to replace all the occurrences of `X` by $\overline{\texttt{X}}$ in `base_to_higher`$_2$.

```
base_to_higher_k 0 X  ←  (is_zero_k X).
base_to_higher_k M X  ←  (input J σ M),
                          (base_to_higher_k J Y),(succ_k Y X).
```

For the actual simulation, we take the following assumptions about the way the Turing machine operates. At the start of the operation, the first squares on the tape hold the input and the rest are blank. The starting state is s_0 and the machine accepts the input if it goes to state s_{yes} in which case it remains there for the rest of the operation. We will give a brief explanation of each predicate.

We have the predicate `symbol`$_\sigma$ `T X`. This succeeds if during the operation of the machine at the time indicated by `T`, the tape at the position indicated by X contains the symbol σ. We will have to create one such predicate for each possible symbol. The predicate `state`$_s$ `T` succeeds if the machine is at state s at the time indicated by T.

Finally the predicate cursor T X succeeds if the cursor of the machine at "time" T is at "position" X. The following program rules are required for the meaning of predicates to be consistent with the initialization of the machine.

symbol$_{\sigma}$ T X $\leftarrow$ (is_zero$_k$ T),(input Y σ W),
    (base_to_higher$_k$ Y X).
symbol$_{\sqcup}$ T X $\leftarrow$ (is_zero$_k$ T),(base_last Y),
    (base_to_higher$_k$ Y Y$_k$), (less_than$_k$ Y$_k$ X).
state$_{s_0}$ T $\leftarrow$ (is_zero$_k$ T).
cursor T X $\leftarrow$ (is_zero$_k$ T), (is_zero$_k$ X).

We proceed with the transition rules of the Turing machine. The following program rules are created for each possible transition of the Turing Machine. For example the transition: "if the head is in symbol σ and in state s then write symbol σ' and go to state s'" gives us:

symbol$_{\sigma'}$ T X $\leftarrow$ (non_zero$_k$ T), (pred$_k$ T T'), (cursor T' X),
    (state$_s$ T'),(symbol$_{\sigma}$ T' X).
state$_{s'}$ T $\leftarrow$ (non_zero$_k$ T),(pred$_k$ T T'), (cursor T' X),
    (state$_s$ T'),(symbol$_{\sigma}$ T' X).
cursor T X $\leftarrow$ (non_zero$_k$ T),(pred$_k$ T T'), (cursor T' X),
    (state$_s$ T'),(symbol$_{\sigma}$ T' X).

The transition: "If the head is in symbol σ and in state s then go to state s' and move the head right" gives us:

symbol$_{\sigma}$ T X $\leftarrow$ (non_zero$_k$ T),(pred$_k$ T T'),(cursor T' X),
    (state$_s$ T'),(symbol$_{\sigma}$ T' X).
state$_{s'}$ T $\leftarrow$ (non_zero$_k$ T),(pred$_k$ T T'),(cursor T' X'),
    (state$_s$ T'),(symbol$_{\sigma}$ T' X').
cursor T X $\leftarrow$ (non_zero$_k$ T),(pred$_k$ T T'),(cursor T' X'),
    (state$_s$ T'),(symbol$_{\sigma}$ T' X'),(succ$_k$ X' X).

The transition: "If the head is in symbol σ and in state s then go to state s' and move the head left" is similarly to the above:

symbol$_{\sigma}$ T X $\leftarrow$ (non_zero$_k$ T), (pred$_k$ T T'), (cursor T' X),
    (state$_s$ T'), (symbol$_{\sigma}$ T' X').
state$_{s'}$ T $\leftarrow$ (non_zero$_k$ T), (pred$_k$ T T'), (cursor T' X'),
    (state$_s$ T'), (symbol$_{\sigma}$ T' X').
cursor T X $\leftarrow$ (non_zero$_k$ T), (pred$_k$ T T'), (cursor T' X'),
    (state$_s$ T'), (symbol$_{\sigma}$ T' X'), (pred$_k$ X' X).

Now one would assume that these sets of Datalog rules are enough. We still haven't considered what happens to the symbol predicates for the pairs of "time" and "position" that the cursor does not succeed or from the machine's perspective, the cursor is not in that position. The machine leaves those symbols unaltered and we simulate this fact with Datalog as follows. The following rules are called "inertia rules" in literature.

symbol$_{\sigma}$ T X $\leftarrow$ (pred$_k$ T T'),(cursor T' X'),
    (symbol$_{\sigma}$ T' X),(less_than$_k$ X X').
symbol$_{\sigma}$ T X $\leftarrow$ (pred$_k$ T T'),(cursor T' X'),
    (symbol$_{\sigma}$ T' X),(less_than$_k$ X' X).

We end the program with the rule that gives the "accept" predicate its value.

$$\texttt{accept} \quad \leftarrow \quad \texttt{(state}_{S_{\text{yes}}}\ \texttt{last}_k\texttt{).}$$

If we carefully inspect the program given above we can see that there are no cyclical dependencies of the predicates that pass through negation. When an expression is negated in the body it is either a predicate with a type of a lower order than the head predicate, or a predicate of the same order in a few select cases that it is easy to see they don't create cyclical dependencies at all.

However, in contrast to the first-order case in the WFS model of semantics, this does not mean that ground terms will only be evaluated to $\{\mathit{false}, \mathit{true}\}$. If the Turing machine does not accept the input string then the `accept` predicate cannot take the value *true* but it can get the value *undef* due to the higher-order existential variables we allowed. To see this consider the following case.
Assume the following transition exists in the Turing machine: "if the head is in symbol σ and in state S^* then write symbol σ' and go to state S_{yes}". Furthermore, assume that with input string w the machine does indeed reach state S^* at one point during its execution. Then the datalog simulation in the model should evaluate $[\![\texttt{state}_{S^*}\ \texttt{T}\ ]\!]_s =$ *true* for some Herbrand state s. Consider a new Herbrand state s' such as $s'(\texttt{T}) = B_F$ where B_F the least element in the Fitting-ordering for the type of the variable `T`. It must be $[\![\texttt{state}_{S^*}\ \texttt{T}\ ]\!]_{s'} =$ *undef* since outputing the value *false* would be a violation of the Fitting-monotonicity condition since there exists at least one more defined element than B_F that it outputs *true*. Outputing the value *true* would completely nullify any validity of the simulation as then any other element should also force the expression to be true again due to Fitting-monotonicity condition. By the same argument, since there exists a state where s where $[\![\texttt{pred}_k\ \texttt{last}_k\ \texttt{T'}\ ]\!]_{s'} =$ *true* then an s' where $s'(\texttt{T}') = B_F$ should output *undef* for that expression as well. By extending the argument for the rest of the body of the following rule:

$$\texttt{state}_{S_{\text{yes}}}\ \texttt{T} \quad \leftarrow \quad \texttt{(non\_zero}_k\ \texttt{T),(pred}_k\ \texttt{T T'),(state}_s\ \texttt{T'),...}$$

Checking the value of $[\![\texttt{state}_{S^*}\ \texttt{T}\ ]\!]_s$ where $s(\texttt{T}) = [\![\texttt{last}_k]\!]$ it has to be at least *undef* since the higher-order variable `T'` is allowed to take the value B_F forcing the right hand side to evalute to *undef*. This forces the evaluation of the expression "$\texttt{state}_{S_{yes}}$ $\texttt{last}_k$" and thus the evaluation of the `accept` predicate to be *undef*.

CHAPTER 6 POSITIVE DATALOG WITHOUT PARTIAL APPLICATION

This chapter is devoted to the fourth row of the results table.

partial application	negation	h.o. existential variables	order 1	order 2	order 3	$\cdots$	order ∞
NO	NO	X	PTIME	PTIME	PTIME	$\cdots$	PTIME

This row describes two sets of Datalog fragments depending on whether X="YES" or X="NO". For k = 1, these fragments coincide, and therefore they have the same expressiveness. Furthermore, demonstrating that PTIME is the upper bound for any k > 1 for programs allowing higher-order existential variables also bounds the expressiveness of programs without those variables. In Section 6.2, we present Lemma 6.5, which also implies the equivalence of these fragment sets. For this reason, for the remainder of this chapter, we will focus on the more general fragment where X="YES," essentially proving the following row.

partial application	negation	h.o. existential variables	order 1	order 2	order 3	$\cdots$	order ∞
NO	NO	YES	PTIME	PTIME	PTIME	$\cdots$	PTIME

6.1 Theorems

In this section, we present the main theorem for this set of language fragments and the necessary lemmas that we use to prove it.

Theorem 6.1. *For every $k \geq 1$ $\mathcal{H}_k^{\exists}$ captures exactly* PTIME.

Proof. Lemma 6.2 shows that for every $k \geq 1$, $\mathcal{H}_k^{\exists}$ expresses at least PTIME. Corollary 6.4 shows that $\mathcal{H}_k^{\exists}$ can express at most PTIME. □

We present now the necessary lemmas that we need to prove the main theorem.

Lemma 6.2. *Let M be a deterministic Turing machine that decides a language L in* PTIME*. Then for any k, there exists a program* P *that belongs in $\mathcal{H}_k^{\exists}$ that decides L.*

Proof. It follows from the fact that for $k = 1$ which is simple first-order Datalog (first-order Datalog can't have partial application), we already know that it captures exactly PTIME as it is also shown in [1]. Now the semantics used in that paper are positive but we have seen that they are equivalent for the ground terms, to the WFS model with theorem 3.19. Therefore, for any $k \geq 1$ we have that $\mathcal{H}_k^{\exists}$ expresses at least PTIME. $\square$

Lemma 6.3. *Let* P *be a Datalog program in* $\mathcal{H}_k^{\exists}$ *with* $k \geq 1$. *Let* $\mathcal{M}_{\mathsf{P}\cup\mathsf{D}_{\mathsf{in}}}$ *be the WFS model of* $\mathsf{P} \cup \mathsf{D}_{in}$ *for a set of input (first-order) relations* D_{in}. *Then there exists an algorithm (or a Turing machine) that when given* D_{in} *as its input, for any propositional or first-order predicate* p *in* P, *it calculates* $\mathcal{M}_{\mathsf{P}\cup\mathsf{D}_{\mathsf{in}}}(\mathsf{p})$. *Furthermore, it runs in polynomial time with respect to* $|\mathsf{D}_{in}|$.

The rest of the chapter is ultimately devoted to the proof of this Lemma which gives us the following corollary result.

Corollary 6.4. *Let* P *be a program in* $\mathcal{H}_k^{\exists}$, $k \geq 1$ *that decides a language L. There exists an algorithm (or a Turing machine) that decides L and runs in time* $O(n^q)$, *where* n *is the length of the input string and* q *is a constant that depends only on* P.

Proof. It follows by considering that we can decide L by transforming any string w to D_w in polynomial time and then run the polynomial algorithm that is given from Lemma 6.3 to find $\mathcal{M}_{\mathsf{P}\cup\mathsf{D}_{\mathsf{in}}}(\mathsf{accept})$. $\square$

From now on in this chapter, we will use the **positive Datalog semantics** for the following analysis of the expressiveness of the language fragments. In chapter 3 we have shown that we can calculate the full meaning of propositional and first-order predicates by using positive semantics since any ground query evaluates to the same truth value under both semantics.

The sections that follow will show the proof of Lemma 6.3 divided into steps. First, given a program P in $\mathcal{H}_k^{\exists}$ we provide a transformation that produces an equivalent program with no higher-order existential variable in the rules. In the positive case we can remove higher-order existential variables by leveraging the monotonicity condition. Then we briefly introduce Bezem semantics and their equivalence to the classical positive semantics that we use in this chapter. We exploit this equivalence to calculate the minimum model under Bezem semantics instead and argue about how we can achieve that in polynomial time for this specific language fragment set and only in the case of calculating the meaning of propositional and first-order predicates.

6.2 A transformation to remove higher-order existential variables from positive Datalog

Consider the following transformation:

Set P' to be a copy of P. Then perform the following changes in P':

- For every type ρ_{V} where $\mathsf{V} : \rho_{\mathsf{V}}$ is a higher order existential variable in P' add a new constant predicate $\mathsf{top}\rho_{\mathsf{V}}$ and the following rule in the program.

$$\mathsf{top}\rho_{\mathsf{V}}\texttt{(X)}.$$

- In every rule of P' replace every instance of an existential variable $\mathsf{V} : \rho_{\mathsf{V}}$ with the constant predicate $\mathsf{top}\rho_{\mathsf{V}}$.

Example 6.1. For example the program

$$\texttt{p} \leftarrow \texttt{r(Q),Q(a)}.$$

becomes

$$\begin{array}{l}\mathsf{top}\rho_{\mathsf{Q}}\texttt{(X)}.\\ \texttt{p} \leftarrow \texttt{r(}\mathsf{top}\rho_{\mathsf{Q}}\texttt{),}\mathsf{top}\rho_{\mathsf{Q}}\texttt{(a)}.\end{array}$$

It is easy to see that the transformation is a polynomial time transformation with respect to the size of the original program P. We will show the following lemma.

Lemma 6.5. *Let* P *in* $\mathcal{H}_k^{\exists}$ *for* $k \geq 2$ *and* P' *the program produced by applying the previous transformation to* P*. The following hold:*

- P' *contains no existential higher-order variables.*
- *For any input relations* D_{in} *let* $\mathcal{M}_{\mathsf{P}\cup\mathsf{D}_{in}}$ *be the minimum model of* $\mathsf{P} \cup \mathsf{D}_{in}$ *and* $\mathcal{M}_{\mathsf{P}'\cup\mathsf{D}_{in}}$ *the minimum model of* $\mathsf{P}' \cup \mathsf{D}_{in}$*. Then for every constant predicate* p *in P it holds that* $\mathcal{M}_{\mathsf{P}\cup\mathsf{D}_{in}}(\mathsf{p}) = \mathcal{M}_{\mathsf{P}'\cup\mathsf{D}_{in}}(\mathsf{p})$*.*

Before proceeding with the proof we assume for convenience that P also includes these top predicates without them replacing any existential higher-order variables in the rules. That is just so we can consider every interpretation of P also as an interpretation of P' without having to extend it. We show that the models of both programs are the same therefore they share the same minimum (positive semantics) model.

Lemma 6.6. *For every Herbrand interpretation* $\mathcal{I}$*,* $\mathcal{I}$ *is a model of* P *if and only if* $\mathcal{I}$ *is a model of* P'*.*

Proof. We show that if an interpretation $\mathcal{I}$ is not a model of P it is not a model of P' and vice versa. We will consider only interpretations that give these new top predicates the meaning of the correct type top element, or else it is trivial that the considered interpretation is not a model for both programs.

Let $\mathcal{I}$ be an interpretation that is not a model of P. Then there exists a rule in P

$$\mathsf{p}\,\mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{E}_1 \wedge \cdots \wedge \mathsf{E}_m$$

and a Herbrand interpretation s such that for all $i \in \{1, \ldots, m\}$, $[\![\mathsf{E}_i]\!]_s(\mathcal{I}) = \mathit{true}$ and $[\![\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n]\!]_s(\mathcal{I}) = \mathit{false}$. The set $\{\mathsf{V}_1, \cdots, \mathsf{V}_n\}$ is the set of formal parameters of the rule and assume $\{\mathsf{V}_{n+1}, \ldots, \mathsf{V}_{n+k}\}$ is the set of existential variables that appear in the body. Now in P', by considering the former transformation there has to be a corresponding rule:

$$\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{E}'_1 \wedge \cdots \wedge \mathsf{E}'_m$$

We show that there exists a Herbrand state s' such that for all $i \in \{1, \ldots, m\}$, $[\![\mathsf{E}'_i]\!]_s(\mathcal{I}) =$ *true* and $[\![\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n]\!]_s(\mathcal{I}) =$ *false*. Consider s' to be one that $s'(\mathsf{V}_i) = s(\mathsf{V}_i) = d_i$ for $i \in \{1, \ldots, n\}$. Then obviously $[\![\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n]\!]_{s'}(\mathcal{I}) =$ *false* by the choice of the interpretation $\mathcal{I}$. For a E'_i we consider cases based on the syntactic form of the expression.

- $\mathsf{E}'_i ==$ "$(\mathsf{top}\rho_{\mathsf{V}'_i}\ \ldots)$" for some $\mathsf{top}\rho_{\mathsf{V}'_i}$ we added. Then it is $[\![\mathsf{E}'_i]\!]_{s'}(\mathcal{I}) =$ *true* since we consider $\mathcal{I}$ where top predicates get the meaning of the corresponding top element.

- $\mathsf{E}'_i ==$ "$(\mathsf{q}\ arg'_1 \ldots arg'_k)$" for some constant predicate q. The corresponding E_i in program P is of the form $\mathsf{E}_i ==$ "$(\mathsf{q}\ arg_1 \ldots arg_k)$" . For every $k \in 1, \cdots, t$ we have

  - If $arg'_k ==$ "V_i" with $0 < i \leq n$ which makes V_i a formal variable of the rule then

    $$[\![arg'_k]\!]_{s'}(\mathcal{I}) = s'(\mathsf{V}_i) = d_i = s(\mathsf{V}_i) = [\![arg_k]\!]_s(\mathcal{I})$$

  - If $arg'_k ==$ "$\mathsf{top}\rho_{\mathsf{V}_i}$" where $i > n$. This means that the argument is a replaced higher-order variable with a top predicate. Since we consider only interpretations that give these predicates the meaning of top elements we have:

    $$[\![arg'_k]\!]_{s'}(\mathcal{I}) = \mathsf{top}\rho_{\mathsf{V}'_i} \geq_{\rho_{\mathsf{V}'_i}} s(\rho_{\mathsf{V}'_i}) = [\![arg_k]\!]_s(\mathcal{I})$$

  - If $arg'_k ==$ "r" for some constant predicate r then again we have that

    $$[\![arg'_k]\!]_{s'}(\mathcal{I}) = [\![r]\!]_{s'}(\mathcal{I}) = [\![r]\!]_s(\mathcal{I}) = [\![arg_k]\!]_s(\mathcal{I})$$

  So for all sub-cases for the arguments, we have that

  $$[\![arg'_k]\!]_{s'}(\mathcal{I}) \geq [\![arg_k]\!]_s(\mathcal{I})$$

  Now since $\mathcal{I}(\mathsf{q})$ is a monotone function we have that:

  $$[\![\mathsf{E}'_i]\!]_{s'}(\mathcal{I}) = [\![(\mathsf{q}\ arg'_1, \ldots, arg'_k)]\!]_{s'}(\mathcal{I}) = [\![\mathsf{q}]\!]_{s'}(\mathcal{I})([\![arg'_1]\!]_{s'}(\mathcal{I}) \ldots [\![arg'_k]\!]_{s'}(\mathcal{I}))$$

  $$\geq_{\rho_q}$$

  $$[\![\mathsf{q}]\!]_s(\mathcal{I})([\![arg_1]\!]_s(\mathcal{I}) \ldots [\![arg_k]\!]_s(\mathcal{I})) = [\![(\mathsf{q}\ arg_1 \ldots arg_k)]\!]_s(\mathcal{I}) = [\![\mathsf{E}_i]\!]_s(\mathcal{I})$$

  In compact form

  $$[\![\mathsf{E}'_i]\!]_{s'}(\mathcal{I}) \geq [\![\mathsf{E}_i]\!]_s(\mathcal{I}) = \textit{true}$$

- $\mathsf{E}'_i ==$ "$(\mathsf{V}_i\ arg'_1, \cdots, arg'_k)$" for some formal higher-order variable V_i. Same analysis as previous gives us $[\![\mathsf{E}'_i]\!]_{s'}(\mathcal{I}) =$ *true*. Instead of a constant monotone predicate, we have at the root $s'(\mathsf{V}_i) = s(\mathsf{V}_i) = d_i$ where d_i is monotone so we can follow the same line of thought as before.

- E'_i == "$(\mathsf{E}_1 \approx \mathsf{E}_2)$". Each one of $\mathsf{E}_1, \mathsf{E}_2$ is either a variable or a constant of type ι. The meaning of the expression under current instantiation is the same regardless of interpretations.

This concludes the argument that if $\mathcal{I}$ is not a model of P it is also not a model of P'.

For the other direction let $\mathcal{I}$ be an interpretation that is not a model of P'. Then there exists a rule in P'

$$\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{E}'_1 \wedge \cdots \wedge \mathsf{E}'_m$$

and a Herbrand interpretation s' such that for all $i \in \{1, \ldots, m\}$, $[\![\mathsf{E}_i]\!]_{s'}(\mathcal{I}) = \mathit{true}$ and $[\![\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n]\!]_s(\mathcal{I}) = \mathit{false}$. By considering the corresponding rule in P

$$\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{E}_1 \wedge \cdots \wedge \mathsf{E}_m$$

and taking s such that for any formal variable V_i $(i \leq n)$ it is $s(\mathsf{V}_i) = s'(\mathsf{V}_i) = d_i$ and for any non formal V'_i it is $s(\mathsf{V}'_i) = d'_i$ where d'_i is the top element for the type $\rho_{\mathsf{V}'_i}$, it is easy to show that for every E_i we also get

$$[\![\mathsf{E}'_i]\!]_s(\mathcal{I}) = \mathit{true}$$

while $[\![\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n]\!]_s(\mathcal{I}) = \mathit{false}$. Therefore, $\mathcal{I}$ is not a model of P. □

Lemma 6.5 allows us to transform every program P in $\mathcal{H}_k^{\exists}$ to one in a more restricted fragment which does not allow existential variables in the body of the rules namely a program in $\mathcal{H}_k$. Furthermore, this transformation depends purely on program P so trivially polynomial to the input size of the input database. We will now briefly describe the Bezem semantics and how we can use an equivalence theorem between that and the semantics of our language (Wadge) to calculate the meaning of all propositional and first-order predicates in the minimum model of P.

6.3 Bezem Semantics and an Equivalence Theorem

We will describe here a new model for semantics that is based on a more syntactical approach. This semantics is defined for a language that is more general than our own. It can easily be verified that our language is a subset of the one used in [3]. It can also be verified that the programs in the fragment $\mathcal{H}_k$ and by former arguments $\mathcal{H}_k^{\exists}$ are also valid programs (named "hoapata" programs in the paper) that we can use Bezem semantics to give them meaning.

Before giving the proper definition, consider the following example of a program in $\mathcal{H}_k$.

Example 6.2.

```
q a.
q b.
p Q ← (Q a).
id R X ← (R X).
```

In Bezem's approach, we take the ground instantiation of a program, which can be viewed as a propositional program that is produced by the original by replacing every variable with all possible appropriately typed terms of the Herbrand universe. Those terms are created using only predicate and individual constants that appear in the program. For example, the ground instantiation of the example program above would be:

Example 6.3.

```
q a.
q b.
p q ← (q a).
id q a ← (q a).
p (id q) ← ((id q) a).
id (id q) a ← ((id q) a).
p (id (id q)) ← ((id (id q)) a).
            ...
```

Notice how each variable is replaced with any appropriately typed term including those that are produced by partial application like `(id (id q))` which has the same type as the constant predicate `q`. This generally gives us a ground instantiation that is infinite. We treat this as an infinite propositional program which implies that we can use the standard fixed point theory of classical logic programming to find which atoms are true at the minimum model.

Now we present the formal definitions and the main theorem we will use without proof. Proof of the theorem can be found in [3].

Definition 6.1. For a program P, we define the *Herbrand universe* for every argument type ρ, denoted by $U_{\mathsf{P},\rho}$, to be the set of all ground terms of type ρ that can be formed out of the individual constants and predicate constants in the program.

The notion of the ground instantiation is given by the following definitions.

Definition 6.2. A *ground substitution* θ is a finite set of the form $\{\mathsf{V}_1/\mathsf{E}_1, \dots, \mathsf{V}_n/\mathsf{E}_n\}$ where the V_i's are different argument variables and each E_i is a ground term having the same type as V_i. We write $dom(\theta) = \{\mathsf{V}_1, \dots, \mathsf{V}_n\}$ to denote the domain of θ.

Definition 6.3. Let θ be a ground substitution and E be an expression. Then, $\mathsf{E}\theta$ is an expression obtained from E as follows:

- $\mathsf{E}\theta = \mathsf{E}$ if E is a predicate constant or individual constant;
- $\mathsf{V}\theta = \theta(\mathsf{V})$ if $\mathsf{V} \in dom(\theta)$; otherwise, $\mathsf{V}\theta = \mathsf{V}$;
- $(\mathsf{E}_1\ \mathsf{E}_2)\theta = (\mathsf{E}_1\theta\ \mathsf{E}_2\theta)$;
- $(\mathsf{E}_1 \approx \mathsf{E}_2)\theta = (\mathsf{E}_1\theta \approx \mathsf{E}_2\theta)$.

If θ is a ground substitution such that *vars*$(\mathsf{E}) \subseteq$ *dom*(θ), then the ground expression $\mathsf{E}\theta$ is called a *ground instance* of E.

Definition 6.4. Let P be a program. A *ground instance of a clause* $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{E}_1, \dots, \mathsf{E}_m$ of P is a formula $(\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n)\theta \leftarrow \mathsf{E}_1\theta, \dots, \mathsf{E}_m\theta$, where θ is a ground substitution whose domain is the set of all variables that appear in the clause, such that for every $\mathsf{V} \in dom(\theta)$ with $\mathsf{V} : \rho$, it is $\theta(\mathsf{V}) \in U_{\mathsf{P},\rho}$. The *ground instantiation of a program* P, denoted by $\mathsf{Gr}(\mathsf{P})$, is the (possibly infinite) set that contains all the possible ground instances of the clauses of P.

Definition 6.5. Let P be a program and let $\mathsf{Gr}(\mathsf{P})$ be its ground instantiation. A *Herbrand interpretation I of* $\mathsf{Gr}(\mathsf{P})$ is defined as a subset of $U_{\mathsf{P},o}$ by the usual convention that, for any $\mathsf{A} \in U_{\mathsf{P},o}$, $I(\mathsf{A}) = true$ iff $\mathsf{A} \in I$. We also extend the interpretation I for every $(\mathsf{E}_1 \approx \mathsf{E}_2)$ ground atom as follows: $I(\mathsf{E}_1 \approx \mathsf{E}_2) =$ *true* if $\mathsf{E}_1 = \mathsf{E}_2$ and *false* otherwise.

Finally, we need the immediate consequence operator.

Definition 6.6. The *immediate consequence operator for* $\mathsf{Gr(P)}$, $\mathcal{T}_{\mathsf{Gr(P)}}$ is the following operator:

$$\mathcal{T}_{\mathsf{Gr(P)}}(I)(\mathsf{A}) = \begin{cases} \textit{true}, & \text{if there exists a clause } \mathsf{A} \leftarrow \mathsf{E}_1, \ldots, \mathsf{E}_m \text{ in } \mathsf{Gr(P)} \\ & \text{such that } I(\mathsf{E}_i) = true \text{ for all } i \in \{1, \ldots, m\} \\ \textit{false}, & \text{otherwise.} \end{cases}$$

We have that the least fixed point of $\mathcal{T}_{\mathsf{Gr(P)}}$ exists and is the minimum model of $\mathsf{Gr(P)}$. We will denote this model as $\mathcal{M}_{\mathsf{Gr(P)}}$. The main theorem we will now present states that with respect to the ground atoms the Bezem approach and our semantics (Wadge) attribute the same truth values. Its proof can be found in [3].

Theorem 6.7 **([3]).** *Let* P *be a program and let* $\mathsf{Gr(P)}$ *be its ground instantiation. Let* M_{P} *be the* $\leq_{\mathcal{I}_{\mathsf{P}}}$*-minimum Herbrand model of* P *and let* $\mathcal{M}_{\mathsf{Gr(P)}}$ *be the* $\leq$*-minimum model of* $\mathsf{Gr(P)}$. *Then, for every* $\mathsf{A} \in U_{\mathsf{P},o}$ *it holds* $[\![\mathsf{A}]\!](M_{\mathsf{P}}) = \mathcal{M}_{\mathsf{Gr(P)}}(\mathsf{A})$.

6.4 Finding (partially) the minimum model of programs in $\mathcal{H}_k^{\exists}$

We can now use the results given in the previous two sections to proceed with the main Lemma 6.3 of this chapter. Ultimately our goal is to use Bezem semantics to find the truth values of propositional and first-order predicates in the minimum model. We also want to achieve that in polynomial time with respect to the input. As it has been presented so far, trying to calculate the minimum model under Bezem semantics is not feasible due to the possibly infinite size of the ground instantiation. We need one last step before we can use the semantics. Consider the following two definitions.

Definition 6.7. Let P be a program and $U^*_{\mathsf{P},o}$ be the subset of $U_{\mathsf{P},o}$ that contains every ground term A such that A does not contain a partially applied expression in its syntactical tree. More specifically A is a ground expression E such that:

- $\mathsf{E} ==$ "$\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n$" for some n-arity constant predicate p where each E_i is either a predicate constant or an individual constant.
- "$c_1 \approx c_2$" for c_1, c_2 some constants of type ι.

Definition 6.8. Let P be a program and $\mathsf{Gr}^*(\mathsf{P})$ be a subset of $\mathsf{Gr(P)}$ that contains each ground instance of a rule

$$\mathsf{A} \leftarrow \mathsf{E}_1, \ldots, \mathsf{E}_m$$

such that $\mathsf{A}, \mathsf{E}_1, \cdots, \mathsf{E}_m \in U^*_{\mathsf{P},o}$.

Example 6.4. The $\mathsf{Gr}^*(\mathsf{P})$ of the program of the example 6.2

```
q a.
q b.
p q ← (q a).
id q a ← (q a).
id q b ← (q b).
```

We can show now the following lemma for programs in the fragment $\mathcal{H}_k$.

Lemma 6.8. *Let* P *be a program in* $\mathcal{H}_k$ *for any* $k \geq 1$*. Let* $\mathcal{M}_{\mathsf{Gr(P)}}$ *be the* $\leq$*-minimum model of* $\mathsf{Gr(P)}$ *and* $\mathcal{M}_{\mathsf{Gr^*(P)}}$ *be the* $\leq$*-minimum model of* $\mathsf{Gr^*(P)}$*. Then for every* $\mathsf{A} \in U^*_{\mathsf{P},o}$

$$\mathcal{M}_{\mathsf{Gr^*(P)}}(\mathsf{A}) = \mathcal{M}_{\mathsf{Gr(P)}}(\mathsf{A})$$

Proof. It follows easily by considering the two important restrictions of programs in fragment $\mathcal{H}_k$ (notice that also $\mathcal{H}_k^{\exists}$ reduces to $\mathcal{H}_k$ after we apply the transformation of lemma 6.5). Specifically they cannot contain higher-order existential variables and partial application in the body of the rules. Consider a term $\mathsf{A} \in U^*_{\mathsf{P},o}$. For A to appear as head in a ground instance of a rule such as:

$$\mathsf{A} \leftarrow \mathsf{E}_1, \dots, \mathsf{E}_m$$

there must be a ground substitution which assigns each variable to either a constant or a constant predicate and is applied to a rule in the program P. By inspecting the syntax of the body of each literal in said rule it is easy to verify that $\{\mathsf{E}_1, \dots, \mathsf{E}_m\}$ must also belong to $U^*_{\mathsf{P},o}$ therefore the ground instance belongs to $\mathsf{Gr^*(P)}$.

By considering each step of the operator $\mathcal{T}_{\mathsf{Gr(P)}}$ we see that for every $\mathsf{A} \in U^*_{\mathsf{P},o}$ we only need to consider ground instances that belong to $\mathsf{Gr^*(P)}$. Therefore, we can find $\mathcal{M}_{\mathsf{Gr(P)}}(\mathsf{A})$ by operating only on $\mathsf{Gr^*(P)}$ and not materializing the possibly infinite $\mathsf{Gr(P)}$. This can be seen equivalently as finding $\mathcal{M}_{\mathsf{Gr^*(P)}}(\mathsf{A})$. □

Notice here how both $\mathsf{Gr^*(P)}$ and $U^*_{\mathsf{P},o}$ are polynomial in size with respect to the size of P and that they can be produced easily in polynomial time with respect to the program. It is also a well-known result that the minimum model of propositional programs can be calculated in polynomial time with respect to their size.

We are now ready to prove Lemma 6.3.

Proof. Take a program P in $\mathcal{P}_k$, $k \geq 1$ and a set of first-order input relations D_{in}. Perform the following actions:

- Apply the transformation given in Lemma 6.5 to get a new equivalent program P' that has no higher-order existential variables. The running time of the transformation is polynomial on P.
- Calculate $\mathsf{Gr^*(P' \cup D_{in})}$ and $U^*_{\mathsf{P' \cup D_{in}},o}$ in running time $O(|P' \cup \mathsf{D}_{in}|^q)$ for some fixed q.
- Find the minimum model of the propositional program that is $\mathsf{Gr^*(P' \cup D_{in})}$. This takes polynomial time with respect to $|\mathsf{Gr^*(P' \cup D_{in})}|$. Given that the bound of $|\mathsf{Gr^*(P' \cup D_{in})}|$ is polynomial with respect to $|P' \cup \mathsf{D}_{in}|$ we get running time polynomial with respect to $|P' \cup \mathsf{D}_{in}|$.

For any propositional predicate p in P' we can retrieve its meaning from $\mathcal{M}_{\mathsf{Gr^*(P \cup D_{in})}}$. We can do the same in polynomial time for first-order constant predicates after we retrieve the truth values of all ground terms that are produced by replacing variables with individual constants. Lemma 6.8 and the equivalence Theorem 6.7 validates that

$$[\![\mathsf{A}]\!]\mathcal{M}_{\mathsf{Gr^*(P' \cup D_{in})}} = [\![\mathsf{A}]\!]\mathcal{M}_{\mathsf{Gr(P' \cup D_{in})}} = [\![\mathsf{A}]\!]\mathcal{M}_{\mathsf{P' \cup D}_{in}}$$

and also by Lemma 6.5

$$[\![\mathsf{A}]\!]\mathcal{M}_{\mathsf{P}'\cup\mathsf{D}_{in}} = [\![\mathsf{A}]\!]\mathcal{M}_{\mathsf{P}\cup\mathsf{D}_{in}}$$

Furthermore, by Theorem 3.19 $[\![\mathsf{A}]\!]\mathcal{M}_{\mathsf{P}\cup\mathsf{D}_{in}}$ is the same as it would be if we were to use Well-Founded semantics. The total running time is polynomial in respect to $|P' \cup \mathsf{D}_{in}|$ and by considering P and subsequently P' to be fixed, polynomial in respect to $|\mathsf{D}_{in}|$ $\square$

CHAPTER 7
CONCLUSION AND FUTURE WORK

In this chapter, we will discuss possible topics to explore in the future as well as mention the unresolved questions of this work. More specifically this thesis does not show the following row from the results table.

partial application	negation (WFS)	h.o. existential variables		order 1	order 2	$\cdots$	order ∞
NO	YES	NO		PTIME	PTIME	$\cdots$	PTIME

This fragment set, which includes negation but no partial application or higher-order existential variables, is expected to lie in PTIME. A few obstacles restricted the inclusion of the result, in its most general form, in this work. Still it is interesting to mention different approaches for it. Consider an approach like the one in chapter 6 where we use an equivalence of extensional and syntactical semantics. This does not hold when we introduce the negation under the Well-Founded model of semantics in the higher-order setting.

Assume the following example program and part of its ground instantiation.

Example 7.1. An example program that evaluates differently under Bezem's semantics

```
r a.
p X :- ∼(q p X).
q P X :- ∼(P X).

r a.
p a :- ∼(q p a).
q p a :- ∼(p a).
```

The WFS model will assign (p a) $=$ *false*. The predicate q takes in one step its meaning, regardless of the meaning of the predicate p and there are no cyclical dependencies. Then (p a) drops from *undef* to *false* in the following iteration step of the outer loop. In Bezem's approach where we treat the program as a propositional based on the ground instantiation, we can see that the ground terms produce a cyclical dependence that contains a double negation. It is equivalent to the program:

```
p :- ∼ q.
q :- ∼ p.
```

This leads to both `(p a)` and `(q p a)` taking the value of *undef*. This mismatch makes the previous approach we took in Chapter 6 unusable. In fact, when we allow richer syntax in our programs like partial application and/or o type (boolean domain) as an argument type, Bezem's approach leads to non-extensional models in this setting.

There are two approaches that we are aware of for solving this issue. The first approach involves finding a strictly polynomial time algorithm that generates a propositional program from the original program while preserving the truth values of ground terms. In the positive case, this corresponds to the default behavior of ground instantiation. The second approach, which is more mechanical, leads to a polynomial time algorithm for model calculation. It is based on demonstrating the following.

- The bottom-up algorithm of finding the WFS model converges in polynomial w.r.t. size of the program number of steps, both for the inner loops and the outer loop.
- We can avoid the step of instantiating higher-order variables by a form of lazy-instantiation where we calculate the output of the function as we need and in an out-of-order fashion.

Another area of study involves the exploration of lifting the ordered-database assumption. Specifically, when we defined the decidability of a language L with Datalog, we assumed an input relation of a specific form derived from a string w and also an 'accept' predicate as the output. This input relation implied an ordering of the elements of the universe, specifically the zero-order constants.

This concept can be extended to encompass any set of input relations and any set of output relations, or in other words general database queries, provided there is an ordering relation in place. However, we also know that first-order Datalog with negation, under *stable* semantics not only is problably more expressive than positive Datalog (captures NP) but also has the capability to lift the ordered-database assumption. Therefore, it is intriguing to establish *stable* semantics for higher-order Datalog and examine its expressiveness in relation to the program order.

BIBLIOGRAPHY

[1] Angelos Charalambidis, Christos Nomikos, and Panos Rondogiannis. ``The Expressive Power of Higher-Order Datalog." In: *Theory Pract. Log. Program.* 19.5-6 (2019), pp. 925–940. DOI: `10.1017/S1471068419000279`. URL: `https://doi.org/10.1017/S1471068419000279`.

[2] Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. ``Approximation Fixpoint Theory and the Well-Founded Semantics of Higher-Order Logic Programs." In: *Theory Pract. Log. Program.* 18.3-4 (2018), pp. 421–437. DOI: `10.1017/S1471068418000108`. URL: `https://doi.org/10.1017/S1471068418000108`.

[3] Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. ``Equivalence of two fixed-point semantics for definitional higher-order logic programs." In: *Theor. Comput. Sci.* 668 (2017), pp. 27–42. DOI: `10.1016/j.tcs.2017.01.005`. URL: `https://doi.org/10.1016/j.tcs.2017.01.005`.

[4] Angelos Charalambidis et al. ``Extensional Higher-Order Logic Programming." In: *ACM Trans. Comput. Log.* 14.3 (2013), 21:1–21:40. DOI: `10.1145/2499937.2499942`. URL: `https://doi.org/10.1145/2499937.2499942`.

[5] Evgeny Dantsin et al. ``Complexity and expressive power of logic programming." In: *ACM Comput. Surv.* 33.3 (2001), pp. 374–425. DOI: `10.1145/502807.502810`. URL: `https://doi.org/10.1145/502807.502810`.

[6] Marc Denecker, Maurice Bruynooghe, and Joost Vennekens. ``Approximation Fixpoint Theory and the Semantics of Logic and Answers Set Programs." In: *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Ed. by Esra Erdem et al. Vol. 7265. Lecture Notes in Computer Science. Springer, 2012, pp. 178–194. DOI: `10.1007/978-3-642-30743-0\_13`. URL: `https://doi.org/10.1007/978-3-642-30743-0%5C_13`.

[7] Neil D. Jones. ``The expressive power of higher-order types or, life without CONS." In: *J. Funct. Program.* 11.1 (2001), pp. 5–94. DOI: `10.1017/s0956796800003889`. URL: `https://doi.org/10.1017/s0956796800003889`.

[8] Christos H. Papadimitriou. ``A note the expressive power of Prolog." In: *Bull. EATCS* 26 (1985), pp. 21–22.

[9] Christos H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994, pp. I–XV, 1–523. ISBN: 978-0-201-53082-7.

[10] William W. Wadge. ``Higher-Order Horn Logic Programming." In: *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*. Ed. by Vijay A. Saraswat and Kazunori Ueda. MIT Press, 1991, pp. 289–303.