doi:10.1017/S1471068425100227

The Power of Negation in Higher-Order Datalog

ANGELOS CHARALAMBIDIS and BABIS KOSTOPOULOS
Harokopio University of Athens, Athens, Greece
(*e-mails:* acharal@hua.gr, kostbabis@hua.gr)

CHRISTOS NOMIKOS
University of Ioannina, Ioannina, Greece
(*e-mail:* cnomikos@cs.uoi.gr)

PANOS RONDOGIANNIS
National and Kapodistrian University of Athens, Athens, Greece
(*e-mail:* prondo@di.uoa.gr)

submitted 27 July 2025; revised 27 July 2025; accepted 29 July 2025

Abstract

We investigate the expressive power of Higher-Order *Datalog*$^\neg$ under both the well-founded and the stable model semantics, establishing tight connections with complexity classes. We prove that under the well-founded semantics, for all $k \geq 1$, $(k+1)$-Order *Datalog*$^\neg$ captures $k-\mathsf{EXP}$, a result that holds without explicit ordering of the input database. The proof of this fact can be performed either by using the powerful existential predicate variables of the language or by using partially applied relations and relation enumeration. Furthermore, we demonstrate that this expressive power is retained within a stratified fragment of the language. Under the stable model semantics, we show that $(k+1)$-Order *Datalog*$^\neg$ captures $\mathsf{co}-(k-\mathsf{NEXP})$ using cautious reasoning and $k-\mathsf{NEXP}$ using brave reasoning, again with analogous results for the stratified fragment augmented with choice rules. Our results establish a hierarchy of expressive power, highlighting an interesting trade-off between order and non-determinism in the context of higher-order logic programing: increasing the order of programs under the well-founded semantics can surpass the expressive power of lower-order programs under the stable model semantics.

KEYWORDS: Higher-Order Datalog, descriptive complexity, well-founded semantics, stable model semantics

1 Introduction

The superior expressive power of higher-order functional programing languages with respect to their first-order counterparts has been thoroughly demonstrated by Jones (2001). In logic programing, the first result of this type was established by Charalambidis *et al.* (2019), where it was demonstrated that positive Higher-Order Datalog programs can capture broader complexity classes as their order increases. In particular, it was demonstrated that for all $k \geq 1$, $(k+1)$-Order Datalog captures $k-\mathsf{EXP}$, under the assumption that the input database is *ordered*. The aforementioned result generalized a classical expressibility theorem which states that (first-order) Datalog captures P (Vardi 1982; Papadimitriou 1985; Immerman 1986; Leivant 1989; Grädel 1992), again under the assumption that the input database is ordered. Notice that the ordering assumption underlying the above results, is actually a rather strong one because it allows (even weak) declarative query languages to simulate the ordering of the tape of a Turing machine.

Recently, Bogaerts *et al.* (2024) defined the well-founded and the stable model semantics of Higher-Order Datalog with negation and illustrated its expressive power with non-trivial examples. Remarkably, one such example was the Generalized Geography two-player game, which is a well-known (Lichtenstein and Sipser 1980) PSPACE-complete problem. The examples given in Bogaerts *et al.* (2024) do not seem to require any ordering of the input and solve the corresponding problems declaratively. As noted by Bogaerts *et al.* (2024), such examples indicate "*that higher-order logic programming under the stable model semantics is a powerful and versatile formalism, which can potentially form the basis of novel ASP systems.*" Such systems may be able to cope with demanding problems that arise in combinatorial optimization, game theory, machine learning theory, and so on (see, e.g.,, the discussion in Bogaerts *et al.* (2016); Amendola *et al.* (2019)). The results of Bogaerts *et al.* (2024) trigger the natural question of the exact characterization of the expressive power of Higher-Order Datalog with negation and of whether we can obtain expressibility results that do not rely on the ordering assumption.

In this paper we undertake the formal study of the expressive power of Higher-Order Datalog with negation – which in the rest of the paper is denoted by Higher-Order $Datalog^{\neg}$. The results we obtain indeed demonstrate that negation is a very powerful construct of the language, justifying the increased expressiveness which was conjectured by Bogaerts *et al.* (2024). Our results, which will be explained in detail in the rest of the paper, are presented in Table 1. All the results in the table are new, with the exception of those results concerning programs of order 1 (which are well-known, see e.g., Dantsin *et al.* (2001); Niemelä (2008)). In particular, the main contributions of the paper can be summarized as follows:

- We establish that $(k+1)$-Order $Datalog^{\neg}$ captures $k-\mathsf{EXP}$ under the well-founded semantics. Notably, this result holds without requiring a predefined ordering of the input database: the use of existential predicate variables, facilitates the construction of such an ordering. Furthermore, we show that even in the fragment of the language without existential predicate variables (denoted by HO-$\text{Datalog}^{\neg,\nexists}$ in Table 1), the same result can be achieved through partial applications and an enumeration procedure. Perhaps even more strikingly, the $k-\mathsf{EXP}$ expressibility result also holds for a stratified fragment of $(k+1)$-Order $\text{Datalog}^{\neg,\nexists}$. This last result is especially unexpected, considering that the stratified fragment of classical (first-order) $Datalog^{\neg}$ exhibits strictly lower expressive power than $Datalog^{\neg}$ under the well-founded semantics (Kolaitis 1991).
- We demonstrate that $(k+1)$-Order $Datalog^{\neg}$ captures $\mathsf{co}-(k-\mathsf{NEXP})$ under the stable model semantics using cautious reasoning and $k-\mathsf{NEXP}$ under brave reasoning. As before, these two results hold with or without the presence of existential predicate variables. Additionally, the two results hold within a fragment of $(k+1)$-Order $Datalog^{\neg}$ that consists of programs that have a stratified part together with

Table 1. *Expressive power of Higher-Order Datalog*$^\neg$ *(with no ordering assumption)*

		Order of the program			
Fragment	Semantics	1	2	$\cdots$	$k+1$
HO-Datalog$^\neg$ HO-Datalog$^{\neg,\nexists}$ Stratified HO-Datalog$^{\neg,\nexists}$	Well-Founded	$\subsetneq$ P	EXP	$\cdots$	k-EXP
HO-Datalog$^\neg$ HO-Datalog$^{\neg,\nexists}$ Stratified+Choices HO-Datalog$^{\neg,\nexists}$	Stable (cautious)	co-NP	co-NEXP	$\cdots$	co-(k-NEXP)
HO-Datalog$^\neg$ HO-Datalog$^{\neg,\nexists}$ Stratified+Choices HO-Datalog$^{\neg,\nexists}$	Stable (brave)	NP	NEXP	$\cdots$	k-NEXP

a simple unstratified part of a very specific form (*choice rules*). In other words, we prove that the expressive power, under the stable model semantics, of $(k+1)$-Order *Datalog*$^\neg$ is equivalent to the power of the aforementioned restricted fragment.

- Since it is well-known that $(k-1)-\mathsf{EXP} \subseteq (k-1)-\mathsf{NEXP} \subseteq k-\mathsf{EXP}$ and $(k-1)-\mathsf{EXP} \subseteq \mathsf{co}-((k-1)-\mathsf{NEXP}) \subseteq k-\mathsf{EXP}$, k-Order *Datalog*$^\neg$ programs under the well-founded semantics are at most as powerful as k-Order *Datalog*$^\neg$ programs under the stable model semantics, which, in turn, are at most as powerful as $(k+1)$-Order *Datalog*$^\neg$ programs under the well-founded semantics (under both the brave and cautious reasoning schemes). This observation illustrates an interesting trade-off between order and non-determinism in the context of higher-order logic programing: by increasing the order of our programs while using well-founded semantics, we can surpass the expressive power provided by non-determinism in lower-order programs under the stable model semantics.

The rest of the paper is structured as follows: Section 2 introduces the language we will be studying. Section 3 derives the expressive power of Higher-Order *Datalog*$^\neg$ under the well-founded semantics and Section 4 the power under the stable model semantics. Section 5 presents a semantics-preserving transformation that eliminates existential predicate variables from clause bodies; this transformation implies that our results hold even without the presence of existential predicate variables. Finally, Section 6 concludes the paper giving pointers for future work.

2 Higher-Order Datalog with negation: preliminaries

In this section we define the syntax of Higher-Order *Datalog*$^\neg$. The language uses two base types: o, the Boolean domain, and ι, the domain of data objects. The composite types are partitioned into *predicate* ones (assigned to predicate symbols) and *argument* ones (assigned to parameters of predicates).

Definition 2.1.
Types are either predicate or argument, denoted by π and ρ respectively, and defined as:

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

As usual, the binary operator $\to$ is right-associative. It can be easily seen that every predicate type π can be written in the form $\rho_1 \to \cdots \to \rho_n \to o$, $n \geq 0$ (for $n = 0$ we assume that $\pi = o$).

Definition 2.2.
The alphabet of Higher-Order *Datalog*$^\neg$ consists of: predicate variables of every predicate type π (denoted by capital letters such as $\mathsf{P}, \mathsf{Q}, \ldots$); predicate constants of every predicate type π (denoted by lowercase letters such as $\mathsf{p}, \mathsf{q}, \ldots$); individual variables of type ι (denoted by capital letters such as $\mathsf{X}, \mathsf{Y}, \ldots$); individual constants of type ι (denoted by lowercase letters such as $\mathsf{a}, \mathsf{b}, \ldots$); the equality constant $\approx$ of type $\iota \to \iota \to o$; the conjunction constant $\wedge$ of type $o \to o \to o$; the inverse implication constant $\leftarrow$ of type $o \to o \to o$; and the negation constant `not` of type $o \to o$.

Arbitrary variables (either predicate or individual ones) will be denoted by R.

Definition 2.3.
The expressions and literals of Higher-Order *Datalog*$^\neg$ are defined as follows. Every predicate variable/constant and every individual variable/constant is an expression of the corresponding type; if E_1 is an expression of type $\rho \to \pi$ and E_2 an expression of type ρ then $(\mathsf{E}_1\ \mathsf{E}_2)$ is an expression of type π. Every expression of type o is called an atom. If E is an atom, then E and $(\ \texttt{not}\ \mathsf{E})$ are literals of type o; if E_1 and E_2 are expressions of type ι, then $(\mathsf{E}_1 \approx \mathsf{E}_2)$ and $\texttt{not}\,(\mathsf{E}_1 \approx \mathsf{E}_2)$ are literals of type o.

We will omit parentheses when no confusion arises.

Definition 2.4.
A rule of Higher-Order *Datalog*$^\neg$ is a formula $\mathsf{p}\ \mathsf{R}_1 \cdots \mathsf{R}_n \leftarrow \mathsf{L}_1 \wedge \ldots \wedge \mathsf{L}_m$, where p is a predicate constant of type $\rho_1 \to \cdots \to \rho_n \to o$, $\mathsf{R}_1, \ldots, \mathsf{R}_n$ are distinct variables of types $\rho_1, \ldots, \rho_n$ respectively and the L_i are literals. The literal $\mathsf{p}\ \mathsf{R}_1 \cdots \mathsf{R}_n$ is the head of the rule and $\mathsf{L}_1 \wedge \ldots \wedge \mathsf{L}_m$ is the body of the rule. A program P of Higher-Order *Datalog*$^\neg$ is a finite set of rules.

We will follow the common logic programing practice and write $\mathsf{L}_1, \ldots, \mathsf{L}_m$ instead of $\mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$ for the body of a rule. For brevity reasons, we will often denote a rule as $\mathsf{p}\ \overline{\mathsf{R}} \leftarrow \mathsf{B}$, where $\overline{\mathsf{R}}$ is a shorthand for a sequence of variables $\mathsf{R}_1 \cdots \mathsf{R}_n$ and B represents the body of the rule. By abuse of notation, in the programs that we will write, we will avoid using currying as much as possible and will use tuples instead, a syntax that is more familiar to logic programmers. The tuple syntax can be directly transformed to the curried one by a simple preprocessing. So, for example, instead of succ `Ord` X Y we will write succ(`Ord`,X,Y), instead of the partial application succ `Ord` we will write succ(`Ord`), and so on. More generally, the partial application $\mathsf{p}\ \mathsf{E}_1\ \cdots\ \mathsf{E}_n$ will be written as $\mathsf{p}(\mathsf{E}_1, \ldots, \mathsf{E}_n)$.

The well-founded and the stable model semantics of Higher-Order *Datalog*$^\neg$, were defined in Bogaerts *et al*. (2024). The main idea of the semantics is to interpret higher-order user-defined predicate constants as three-valued relations over two-valued objects, that is as functions that take classical relations as arguments and return *true*, *false*, or *undef*. This interpretation of predicate constants, apart from giving a simple denotation of the various constructs of the language, also allows one to use Approximation Fixpoint Theory (Denecker *et al.* 2000, 2004), in order to define a variety of semantics for the language (such as well-founded, stable, Kripke-Kleene, and so on). For the reader of the main part of the present paper, a deep understanding of this semantics is not necessary: the programs that we give can be understood purely declaratively, without resorting to the help of the semantics (in the same way that a logic programmer does not need to master its model-theoretic semantics in order to write or understand a program). Programs of our language simply define extensional higher-order relations. For example, in the well-founded semantics a unary second-order predicate simply denotes a relation that takes a classical set as an argument and returns *true*, *false*, or *undef* as the result; actually, most of our programs will be *stratified* (see the forthcoming Definition 2.7), which means that they actually denote classical higher-order relations (i.e., they never return *undef* as the result). Moreover, in the stable model semantics all the predicates denote two-valued relations. The only points where the reader will have to delve deeper into the semantics, is in order to understand the proofs of certain theorems provided in an appendix as supplementary material. For this reason (and also due to space restrictions) the full presentation of the semantics is also given as supplementary material.

The notion of *order* of a predicate, is formally defined as follows:

Definition 2.5.
The *order* of a type is recursively defined as follows:

$$\begin{aligned} order(\iota) &= 0 \\ order(o) &= 1 \\ order(\rho \to \pi) &= \max\{order(\rho)+1, order(\pi)\} \end{aligned}$$

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

Definition 2.6.
For all $k \geq 1$, k-Order *Datalog*$^\neg$ is the fragment of Higher-Order *Datalog*$^\neg$ in which all variables have order less than or equal to $k-1$ and all predicate constants in the program have order less than or equal to k.

The following example showcases many aspects of the language and additionally introduces some useful predicates that will be needed in our subsequent simulations.

Example 1.
We define the relation `hamilton(X,Y)` which is true if there exists a Hamilton path from vertex X to vertex Y in a graph represented by a binary predicate e which specifies the edges of the graph. The first rule in the definition of `hamilton` is the following:

```
hamilton(X,Y)←ordering(Ord),first(Ord,X),last(Ord,Y),subset(succ(Ord),e).
```

The above rule states that there exists a Hamilton path from vertex X to vertex Y if there exists a relation Ord that is a strict total ordering with first element X and last element Y and for every two consecutive elements in Ord the corresponding edge exists in e. Notice the use of Ord in the above rule: it is a predicate variable that does not appear in the head of the rule and therefore it is an existentially quantified variable of the body (i.e., the body can be read as "there exists a relation Ord such that …"). To be a strict total ordering, Ord must be irreflexive, transitive, and every two different elements must be related. This can be expressed with the following rules:

```
ordering(Ord)←connected(Ord),transitive(Ord),irreflexive(Ord).
connected(Ord)←not disconnected(Ord).
disconnected(Ord)←not Ord(X,Y),not Ord(Y,X),not(X≈Y).
transitive(Ord)←not non_transitive(Ord).
non_transitive(Ord)←Ord(X,Y),Ord(Y,Z),not Ord(X,Z).
irreflexive(Ord)←not non_irreflexive(Ord).
non_irreflexive(Ord)←Ord(X,X).
```

It is interesting to note above how we can implement universal quantification in the body of a rule: for example, to express the fact that Ord is connected, that is that for all X, Y either X is related to Y or vice-versa, we just require that Ord is not disconnected, that is it is not the case that there exist X, Y that are not related. This is a common trick that we will use throughout the paper in order to represent universal quantification.

We now define the predicates first, last and succ. Predicate first(Ord,X) is true for X being the individual constant that is the first element with respect to the ordering specified by Ord. Likewise, last(Ord,X) is true if X is the last element in Ord. The predicate succ(Ord,X,Y) is true for X and Y that are sequential in Ord.

```
first(Ord,X)←not nfirst(Ord,X).
nfirst(Ord,X)←Ord(Z,X).
last(Ord,X)←not nlast(Ord,X).
nlast(Ord,X)←Ord(X,Y).
succ(Ord,X,Y)←Ord(X,Y),not nsequential(Ord,X,Y).
nsequential(Ord,X,Y)←Ord(X,Z),Ord(Z,Y).
```

Finally, we have the rules for subset:

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

The above two rules use again the trick for implementing universal quantification.

As in the case of first-order logic programs with negation, there exists a simple notion of stratification for higher-order logic programs with negation (Bogaerts *et al*. 2024).

Definition 2.7.
A program P is called stratified if there exists a function S mapping predicate constants to natural numbers, such that for each rule $\mathsf{p}\,\overline{\mathsf{R}} \leftarrow \mathsf{L}_1,\ldots,\mathsf{L}_m$ and any $i \in \{1,\ldots,m\}$:

- $S(\mathsf{q}) \leq S(\mathsf{p})$ for every predicate constant q occurring in L_i.
- If L_i is of the form (`not` E), then $S(\mathsf{q}) < S(\mathsf{p})$ for each predicate constant q occurring in E.

- For any subexpression of L_i of the form $(\mathsf{E}_1\ \mathsf{E}_2)$, $S(\mathsf{q}) < S(\mathsf{p})$ for every predicate constant q occurring in E_2.

A possibly unexpected aspect of the above definition, is the last item, which says that the stratification function should not only increase because of negation, but also because of higher-order predicate application. The intuitive reason for this is that in Higher-Order *Datalog*$^\neg$ one can define a higher-order predicate which is identical to negation, for example, by writing neg P $\leftarrow$ not P. As a consequence, it is reasonable to assume that predicates occurring inside an application of neg should be treated similarly to predicates appearing inside the negation symbol.

One can easily verify that the Hamiltonian Path program of Example 1, is stratified. As we are going to see, the expressive power of stratified programs is the same as that of the non-stratified ones for orders $k \geq 2$ under the well-founded semantics.

Languages such as Higher-Order *Datalog*$^\neg$, are usually referred as *formal query languages*. A program in our language can be considered to compute a query in the following sense: a first-order predicate, like e in Example 1, will be called an *input predicate* and its denotation (as a set of ground atoms) constitutes what is called the *input database*, usually denoted by D_{in}; a first-order predicate like Hamilton in Example 1, will be an *output* one and its denotation constitutes the *output database*, usually denoted by D_{out}.

More formally, a *database schema* σ is a finite set of first-order predicate symbols with associated arities. A *database* over a schema σ is a finite set of ground atoms whose predicate symbols belong to σ. A *query* is a mapping from databases over a schema σ_1 to databases over a schema σ_2. A program P can be seen as a query $\mathcal{Q}_\mathsf{P}$ such that $D_{out} = \mathcal{Q}_\mathsf{P}(D_{in})$. We are interested in queries that are *generic* (Immerman 1986), that is queries that do not depend on the names of the individual constants in the input database. Given a fragment of our language, we are interested in the *expressive power* of the fragment under a given semantics, namely the set of queries that can be defined by programs of the fragment. In particular, we want to demonstrate that such a fragment *captures a complexity class* $\mathcal{C}$, that is it can express *exactly* all the queries whose *evaluation complexity* belongs to $\mathcal{C}$. By evaluation complexity we mean the complexity of checking whether a given atom belongs to the output database. Notice that different semantics of the fragments we study may lead to different evaluation complexities and therefore to capturing different complexity classes. Therefore, our results will be of the form "*the fragment X of Higher-Order Datalog*$^\neg$*, under the Y semantics, captures the complexity class Z.*"

In this work we consider the well-founded semantics, the stable model semantics with cautions reasoning and the stable model semantics with brave reasoning. In particular, under the well-founded semantics, a given atom belongs to the output database if and only if it is true in the well-founded model. Moreover, under the stable model semantics with cautious (resp. brave) reasoning, a given atom belongs to the output database if and only if it is true in all stable models (resp. in at least one stable model).

3 Expressive power under the well-founded semantics

The purpose of this section is to demonstrate that for all $k \geq 1$, $(k+1)$-Order *Datalog*$^\neg$, under the well-founded semantics, captures $k - \mathsf{EXP}$. Proofs of such results usually consist of two parts; in our case these two parts can be intuitively described as follows:

- We show that every Turing machine that takes as input an encoding of an input database and computes a query over this relation that belongs to $k - \mathsf{EXP}$, can be simulated by a $(k+1)$-Order *Datalog*$^\neg$ program (whose meaning is understood under the well-founded semantics).
- We show that computing the well-founded semantics of every $(k+1)$-Order *Datalog*$^\neg$ program over an input database, can be done in k-exponential time with respect to the number n of individual constants in the input database.

In the main part of the paper, we focus on the proof of the first result. The proof of the second result is given in an appendix as supplementary material. The proof of the first result has two important points that have to receive special attention, on which we briefly comment below.

Ordering of the input database: A Turing machine encodes an input database as a string on its tape. Such an encoding provides an implicit strict total ordering of the input. On the other hand, Higher-Order *Datalog*$^\neg$ does not have a notion of tape, and therefore the input database is, at first sight, unordered. In less powerful languages, such as for example the language of Charalambidis *et al*. (2019), this mismatch is solved by imposing an *explicit* strict total ordering on the individual constants of the input database. However, it turns out that in Higher-Order *Datalog*$^\neg$ we do not need this ordering trick (usually referred in the literature as the *ordering assumption*): we can generate an ordering `Ord` of the constants of the input database and then use it to perform the Turing machine simulation.

Representing numbers: Since we want to simulate the operation of a Turing machine, we need to have a numbering scheme in our language in order to count the steps of the Turing machine and the positions on its tape. At first, we need some base-numbers; actually, it is sufficient to use the n individual constants in the input database. The trick here is to use these constants directly as numbers: we generate an ordering `Ord` on these constants and require that it is a strict total order. We then show how to simulate bigger numbers, namely numbers polynomially related to n. Finally, since we simulate the operation of a Turing machine that runs in k-exponential time, we must be able to represent bigger numbers. The trick here is to use higher-order relations: as the order of our programs increases, the more numbers we can represent. Actually, as we demonstrate, $(k+1)$-Order *Datalog*$^\neg$ is sufficient to represent k-exponential numbers.

Regarding the ordering of the input database, we already have all the required machinery ready from Example 1. We can use the predicate ordering to generate a *strict total order* `Ord`, over the constant symbols of the input database, which we will then use in all our simulations. In other words, the constants of our input database play, under this ordering, the role of *base-numbers* in our simulation (i.e., "small" numbers that can get up to $n - 1$). Given an individual constant c of the input database, we will say it represents the natural number m, or more formally $num(\mathsf{c}) = m$, if and only if in the strict total order `Ord` the constant c is the $(m+1)$-th element. In the following, we see how we can use these base-numbers to represent even bigger ones.

3.1 Representing numbers

Representing polynomially-big numbers: To represent natural numbers up to $n^{d+1}-1$, where d is any arbitrary but fixed natural number, we use tuples of individual constants with fixed length of size $d+1$. The following predicates define the "first" and "last" of such numbers (denoted by $\texttt{first}_0$ and $\texttt{last}_0$) and the "less-than" and "successor" relations on them (denoted by $\texttt{lt}_0$ and $\texttt{succ}_0$). Notice that the following definitions use the first and last predicates defined in Example 1.

```
first_0(Ord,X_0,...,X_d)←first(Ord,X_0),...,first(Ord,X_d).
last_0(Ord,X_0,...,X_d)←last(Ord,X_0),...,last(Ord,X_d).
lt_0(Ord,X_0,...,X_d,Y_0,...,Y_d)← Ord(X_d,Y_d).
lt_0(Ord,X_0,...,X_d,Y_0,...,Y_d)← Ord(X_{d-1},Y_{d-1}),(X_d ≈ Y_d).
...
lt_0(Ord,X_0,...,X_d,Y_0,...,Y_d)← Ord(X_0,Y_0),(X_1 ≈ Y_1),...,(X_d ≈ Y_d).
succ_0(Ord,X̄,Ȳ)←lt_0(Ord,X̄,Ȳ),not nsequential_0(Ord,X̄,Ȳ).
nsequential_0(Ord,X̄,Ȳ)← lt_0(Ord,X̄,Z̄),lt_0(Ord,Z̄,Ȳ).
```

In the tuple-representation of numbers, tuple ($\texttt{X}_0, \ldots,$ $\texttt{X}_d$) of base-elements represents the number $num(\texttt{X}_0, \ldots, \texttt{X}_d) = num(\texttt{X}_0) + num(\texttt{X}_1)\cdot n + \cdots + num(\texttt{X}_d)\cdot n^d$.

Representing exponentially-big numbers: We now demonstrate how we can represent "exponentially-big" numbers as higher-order relations. In the foregoing discussion we will need the following notation: $\exp_0(x) = x$ and $\exp_{n+1}(x) = 2^{\exp_n(x)}$. Let $N_0 = n^{d+1}-1$ be the largest number that can be represented by $(d+1)$-tuples of individual constants and for $k \geq 1$, let N_k be the largest number that can be represented by using k-order relations. We can exponentially increase the numbers up to the number $N_{k+1} = \exp_1(N_k+1)-1$ by using $(k+1)$-order relations. One can easily see that $N_k = \exp_k(n^{d+1})-1$.

If the k-order relations representing numbers up to N_k are of type ρ, then it suffices to use higher-order relations of type $\rho \to o$ in order to represent numbers up to N_{k+1}. This is essentially a binary representation where the lower order numbers denote bit positions. Formally, let Z be a $(k+1)$-order element and $\texttt{R}_0, \ldots, \texttt{R}_{N_k}$ be the ordering of the elements that represent numbers in the previous counting module. Let f be the function mapping *true* to 1 and *false* to 0. Then we have $num(\texttt{Z}) = f(\texttt{Z}(\texttt{R}_0)) + f(\texttt{Z}(\texttt{R}_1))\cdot 2 + \cdots + f(\texttt{Z}(\texttt{R}_{N_k}))\cdot 2^{N_k}$. We begin with predicates testing for the first and the last number.

```
first_{k+1}(Ord,N)← not nfirst_{k+1}(Ord,N).
nfirst_{k+1}(Ord,N)← N(X).
last_{k+1}(Ord,N)← not nlast_{k+1}(Ord,N).
nlast_{k+1}(Ord,N)← not N(X).
```

The following definitions describe the "less than" relation between two elements that represent numbers. We examine if a number N is less than M by comparing the two numbers bit by bit in their binary representation. The successor of a number is defined with the use of less-than.

```
lt_{k+1}(Ord,N,M)← last_k(Ord,X),bit_{k+1}(Ord,N,M,X).
bit_{k+1}(Ord,N,M,X)← not N(X),M(X).
bit_{k+1}(Ord,N,M,X)← N(X),M(X),succ_k(Ord,Y,X),bit_{k+1}(Ord,N,M,Y).
bit_{k+1}(Ord,N,M,X)← not N(X),not M(X),succ_k(Ord,Y,X),bit_{k+1}(Ord,N,M,Y).
succ_{k+1}(Ord,N,M)← lt_{k+1}(Ord,N,M), not nsequential_{k+1}(Ord,N,M).
nsequential_{k+1}(Ord,N,M)← lt_{k+1}(Ord,N,Z),lt_{k+1}(Ord,Z,M).
```

For $k=0$ the above code is slightly different: variables X and Y should be replaced with $\bar{\texttt{X}}$ and $\bar{\texttt{Y}}$.

3.2 Turing machine simulation

We now demonstrate how any query that belongs to $k-\mathsf{EXP}$ ($k \geq 1$), can be expressed by a $(k+1)$-Order *Datalog*$^\neg$ program under the well-founded semantics. It suffices to assume that the output schema of the query consists of a single output predicate, since every query can be decomposed into multiple queries of this form. Since the query belongs to $k-\mathsf{EXP}$, there exists a Turing machine that given on its tape an input database under some sensible encoding, decides whether a tuple belongs to the output relation of the query in at most $\exp_k(n^d)$ steps, where n is the number of constant symbols in the input database and d is some constant. We simulate this Turing machine with a $(k+1)$-Order *Datalog*$^\neg$ program.

Encoding the input: Before presenting the simulation of the Turing machine M, we mention certain simplifying assumptions, which do not affect the generality of the subsequent results.

- The input database consists of a single binary relation in and the output database is also a single binary relation out. In the following, the number of constants in the input database is denoted by n.
- The alphabet of M that will be simulated is $\Sigma = \{0, 1, \square\}$. M expects the input relation in as the standard binary encoding of a graph, which is based on the ordering of the individual constants, in the first n^2 cells of its tape. For example, if the pair (x, y) belongs to in, then the tape of M contains a "1" at cell position $num(x) + num(y) \cdot n$, otherwise it contains "0."
- M decides whether a tuple (a, b) belongs to the output relation out. The next n^2 cells of its tape are used to encode (a, b). All these cells contain the symbol "0," except for the cell at position $num(a) + num(b) \cdot n + n^2$ which contains "1."
- M reaches its accepting state *yes* if and only if the tuple (a, b) belongs to the output relation out.

The following two predicates encode the binary input relation in and the tuple (a, b) as a binary string.

```
input_1(A,B,Ord,X,Y,Z_2,...,Z_d)← first(Ord,Z_2),...,first(Ord,Z_d),in(X,Y).
input_0(A,B,Ord,X,Y,Z_2,...,Z_d)← first(Ord,Z_2),...,first(Ord,Z_d),not in(X,Y).
input_1(A,B,Ord,Z_0,Z_1,Z_2,...,Z_d)← (Z_0 ≈ A),(Z_1 ≈ B),first(Ord,Z),succ(Ord,Z,Z_2),
                                      first(Ord,Z_3),...,first(Ord,Z_d).
input_0(A,B,Ord,Z_0,Z_1,Z_2,...,Z_d)← not(Z_0 ≈ A),first(Ord,Z),succ(Ord,Z,Z_2),
                                      first(Ord,Z_3),...,first(Ord,Z_d).
input_0(A,B,Ord,Z_0,Z_1,Z_2,...,Z_d)← not(Z_1 ≈ B),first(Ord,Z),succ(Ord,Z,Z_2),
                                      first(Ord,Z_3),...,first(Ord,Z_d).
```

The predicate input$_1$(A,B,Ord,Z$_0$,. . .,Z$_d$) is true if the symbol 1 will be written during the initialization of M in the cell of the tape represented by the number (Z$_0$,. . .,Z$_d$). Similarly, input$_0$(A,B,Ord,Z$_0$,. . .,Z$_d$) is true if the symbol 0 will be written in that position.

Initial configuration of the Turing Machine: In order to represent the configurations of the Turing machine we use a higher-order predicate for each state and symbol, and a higher-order predicate for the cursor position. The predicate state$_s$(A,B,Ord,T) is true if at time T the Turing machine is in state s. The predicate symbol$_\sigma$(A,B,Ord,T,P), where $\sigma \in \{0, 1, \Box\}$, is true if at time T the tape has symbol σ written in position P. The predicate cursor(A,B,`Ord`,T,P) is true if at time T the cursor is in position P.

We also need a higher-order predicate to lift the tuple representation of numbers to the k-order numbering notation. Predicate lift$_k$(`Ord`,$\bar{\texttt{X}}$,M) transforms the number represented by the tuple $\bar{\texttt{X}}$ in the zero-order notation to the same number M in the k-order notation.

```
lift_k(Ord,X̄,M)←first_0(Ord,X̄),first_k(Ord,M).
lift_k(Ord,X̄,M)←succ_0(Ord,Z̄,X̄),succ_k(Ord,M′,M),lift_k(Ord,Z̄,M′).
```

We proceed to describe the initial configuration of the Turing machine. At time 0, the machine is in its initial state s_0, the tape contains only the binary string of the encoded input and all the other positions are filled with the symbol $\Box$, and the cursor is in position 0. This is described by the following rules:

```
state_s0(A,B,Ord,T)←first_k(Ord,T).
cursor(A,B,Ord,T,P)←first_k(Ord,T),first_k(Ord,P).
symbol_0(A,B,Ord,T,P)←first_k(Ord,T),input_0(A,B,Ord,X̄),lift_k(Ord,X̄,P).
symbol_1(A,B,Ord,T,P)←first_k(Ord,T),input_1(A,B,Ord,X̄),lift_k(Ord,X̄,P).
symbol_□(A,B,Ord,T,P)←first_k(Ord,T),not symbol_0(A,B,Ord,T,P),
                      not symbol_1(A,B,Ord,T,P).
```

We now specify how the execution of the Turing machine is simulated.

Simulating transitions: To describe the transitions of the machine, we create rules which assert the next configuration of the machine, based on the current one and the transition function. Let $(s, \sigma) \rightarrow (s', \sigma', \text{right})$ be a transition, indicating that if the current state of the machine is s and its cursor reads the symbol σ, then the machine changes its state to s', it writes σ' on the current cursor position and then moves the cursor to the right. In our simulation we use an auxiliary predicate current$_{(s,\sigma)}$(A,B,`Ord`,T,P) which is true if at the moment T the state of the machine is s, the cursor position is P and the cursor reads the symbol σ.

```
current_(s,σ)(A,B,Ord,T,P)←state_s(A,B,Ord,T),cursor(A,B,Ord,T,P),
                           symbol_σ(A,B,Ord,T,P).
```

Now, the transition can be simulated by the following rules:

```
state_s′(A,B,Ord,T′)←current_(s,σ)(A,B,Ord,T,P),succ_k(Ord,T,T′).
symbol_σ′(A,B,Ord,T′,P)←current_(s,σ)(A,B,Ord,T,P),succ_k(Ord,T,T′).
cursor(A,B,Ord,T′,P′)←current_(s,σ)(A,B,Ord,T,P),
                      succ_k(Ord,P,P′),succ_k(Ord,T,T′).
```

Other types of transitions can be expressed in a similar way. We also add what is commonly called "inertia rules." They ensure that every position of the tape except for the position of the cursor, retains its content. For any symbol σ we include the following:

```
symbol_σ(A,B,Ord,T′,P′)←succ_k(Ord,T,T′),symbol_σ(A,B,Ord,T,P′),
                        cursor(A,B,Ord,T,P),lt_k(Ord,P,P′).
symbol_σ(A,B,Ord,T′,P′)←succ_k(Ord,T,T′),symbol_σ(A,B,Ord,T,P′),
                        cursor(A,B,Ord,T,P),lt_k(Ord,P′,P).
```

Finally, in order to produce the output relation, we use the following:

```
out(A,B)←ordering(Ord),state_yes(A,B,Ord,T).
```

The out predicate is the one that "initiates the simulation": it produces, using an existential predicate variable, the ordering `Ord` that is used throughout the simulation and it verifies that there exists a value T that represents a time point (within the range of representable numbers) such that the machine reaches an accepting state. The above simulation leads to the following theorem:

Theorem 1.
Every query in $k-\mathsf{EXP}$ ($k \geq 1$) can be expressed by a $(k+1)$-Order *Datalog*$^\neg$ program.

The opposite direction of the above theorem also holds, as stated by the following theorem.

Theorem 2.
Let P be a $(k+1)$-Order *Datalog*$^\neg$ program that defines a query $\mathcal{Q}_\mathsf{P}$ under the well-founded semantics. Then, there exists a deterministic Turing machine that takes as input an encoding of a database D that uses n individual constant symbols and a ground atom $p(\bar{a})$, where p is a predicate constant of P and $\bar{a}$ is a tuple of those individual constants, and decides whether $p(\bar{a}) \in \mathcal{Q}_\mathsf{P}(D)$, in at most $exp_k(n^d)$ steps for some constant d.

By inspecting the simulation program, one easily sees that it is stratified. Therefore, we get the following result:

Corollary 3.1.
$(k+1)$-Order *Datalog*$^\neg$ and Stratified $(k+1)$-Order *Datalog*$^\neg$ capture $k-\mathsf{EXP}$ under the well-founded semantics.

As a consequence, Higher-Order *Datalog*$^\neg$ and Stratified Higher-Order *Datalog*$^\neg$ both capture $\mathsf{ELEMENTARY}$ (i.e., the union of $k-\mathsf{EXP}$ for all k).

4 Expressive power under the stable model semantics

In this section, we study the expressiveness of Higher-Order *Datalog*$^\neg$ under the stable model semantics. We demonstrate how any query that belongs to $\mathsf{co}-(k-\mathsf{NEXP})$ ($k \geq 1$), can be expressed by a $(k+1)$-Order *Datalog*$^\neg$ program under the stable model semantics with cautious reasoning. Since the query belongs to $\mathsf{co}-(k-\mathsf{NEXP})$, there exists a non-deterministic Turing machine M that decides whether a tuple belongs to the complement of the output relation of the query in at most $\exp_k(n^d)$ steps, where n is the number of constants in the input database and d is a constant. Without loss of generality, we assume that each computational path of M terminates after at most $\exp_k(n^d)$ steps at a state in {yes, no}. We simulate M with a $(k+1)$-Order *Datalog*$^\neg$ program.

For the most part, the simulation is the same as that of Section 3. It is intuitively helpful to consider each stable model of the following simulation as a possible computation path the machine could have taken. We will add some additional "choice" rules to simulate those non-deterministic transitions at any possible time step.

Let (σ, s) be a pair of a symbol and a state of the Turing machine and assume there exist m possible transitions from this pair. Since the machine is non-deterministic, m can be greater than 1. We add the following predicates and rules to our program.

```
b$_{\sigma,s,m}$(T)← not b$_{\sigma,s,1}$(T),...,not b$_{\sigma,s,m-1}$(T).
b$_{\sigma,s,m-1}$(T)← not b$_{\sigma,s,1}$(T),...,not b$_{\sigma,s,m-2}$(T),not b$_{\sigma,s,m}$(T).
...
b$_{\sigma,s,1}$(T)← not b$_{\sigma,s,2}$(T),...,not b$_{\sigma,s,m}$(T).
```

This ensures that in every stable model and for every time point T, exactly one of $\mathrm{b}_{\sigma,s,i}(\mathrm{T})$, $i \in \{1,\ldots,m\}$, is true.

Let also the transition table be $(\sigma, s) \rightarrow (\sigma'_i, s'_i, \mathrm{move}_i)$ for $i = 1,\ldots,m$. Like in the deterministic case we create rules for each one such transition but we also add the previous branching predicates so that only exactly one rule can be "active" in a stable model.

```
state$_{s'_i}$(A,B,Ord,T')← b$_{\sigma,s,i}$(T),current$_{(s,\sigma)}$(A,B,Ord,T,P),succ$_k$(Ord,T,T').
symbol$_{\sigma'_i}$(A,B,Ord,T',P)← b$_{\sigma,s,i}$(T),current$_{(s,\sigma)}$(A,B,Ord,T,P),succ$_k$(Ord,T,T').
```

Assuming, for example, that move_i is "right" (and likewise for "left" and "stay"):

```
cursor(A,B,T',P')← b$_{\sigma,s,i}$(T),current$_{(s,\sigma)}$(A,B,Ord,T,P),
                     succ$_k$(Ord,T,T'),succ$_k$(Ord,P,P').
```

If every computational path of the Turing machine reaches state "no," then for every stable model there exists a time point T such that state_{no}(A,B,Ord,T) is true. Therefore, the following rule defines the output relation under cautious reasoning:

```
out(A,B)← ordering(Ord),state$_{no}$(A,B,Ord,T).
```

The above discussion leads to the following theorem:

Theorem 3.
Every query in $\mathsf{co}-(k-\mathsf{NEXP})$ can be expressed by a $(k+1)$-Order *Datalog*$^\neg$ program under the stable model semantics and cautious reasoning.

The following theorem is the converse of the previous one and its proof can be found in the supplementary material.

Theorem 4.
Let P be a $(k+1)$-Order *Datalog*$^\neg$ program that defines a query $\mathcal{Q}_\mathsf{P}$ under the stable model semantics and cautious reasoning. Then, there exists a non-deterministic Turing machine that takes as input an encoding of a database D that uses n individual constant symbols and a ground atom $p(\bar{a})$, where p is a predicate constant of P and $\bar{a}$ is a tuple of those individual constants, and decides whether $p(\bar{a}) \notin \mathcal{Q}_\mathsf{P}(D)$, in at most $exp_k(n^d)$ steps for some constant d.

Notice that our simulation consists of a stratified program together with the rules that define the $\mathrm{b}_{\sigma,s,m}$. We call this fragment of Higher-Order *Datalog*$^\neg$ "*Stratified+Choices Higher-Order Datalog*$^\neg$." We therefore have the following result.

Corollary 4.1.
$(k+1)$-Order *Datalog*$^\neg$ and Stratified+Choices $(k+1)$-Order *Datalog*$^\neg$ capture $\mathsf{co}-(k-\mathsf{NEXP})$ under the stable model semantics and cautious reasoning.

By a similar kind of analysis we can derive a result for the stable model semantics under brave reasoning. The arguments are very similar and omitted.

Corollary 4.2.
$(k+1)$-Order *Datalog*$^\neg$ and Stratified+Choices $(k+1)$-Order *Datalog*$^\neg$ capture $k-\mathsf{NEXP}$ under the stable model semantics and brave reasoning.

Since $(k-1)-\mathsf{EXP} \subseteq (k-1)-\mathsf{NEXP} \subseteq k-\mathsf{EXP}$ and $(k-1)-\mathsf{EXP} \subseteq \mathsf{co}-((k-1)-\mathsf{NEXP}) \subseteq k-\mathsf{EXP}$ it follows that Higher-order *Datalog*$^\neg$ has the same expressive power (namely $\mathsf{ELEMENTARY}$) under both the well-founded and the stable model semantics, in both reasoning schemes.

5 Removing higher-order existential predicate variables

The Turing machine simulations discussed in the previous sections rely extensively on existential predicate variables. Actually, the simulations do not need the full fragment of Higher-Order *Datalog*$^\neg$. In this section, we explore whether the same expressive power can be achieved without the use of such variables but utilizing other powerful constructs of the language, namely partially applied predicates. To this end, we introduce a semantics-preserving transformation that converts every $(k+1)$-Order *Datalog*$^\neg$ program containing existential predicate variables of order $k \geq 1$ into an equivalent program of the same order that does not contain existential predicate variables. This result implies that $(k+1)$-Order *Datalog*$^\neg$ without existential predicate variables has the same expressive power as the full language. Furthermore, the transformation we propose preserves stratification: if the original program is stratified, then the transformed program is stratified as well. In other words, stratified programs without existential predicate variables can be as expressive as unstratified programs with existential predicate variables. We first illustrate the proposed transformation with our Hamilton example.

Example 2.
Consider our initial rule for the Hamilton query:

```
hamilton(X,Y)← ordering(Ord),first(Ord,X),last(Ord,Y),subset(succ(Ord),e).
```

The key idea of the transformation is that instead of using an existential predicate variable for finding an appropriate relation Ord that is a strict total order, we can use an iterative procedure that starts from the empty relation and successively adds to it pairs of individual constants until we get a relation that is indeed a strict total order. In other words, we construct ourselves, in a bottom-up way, the strict total order. The corresponding transformed program is the following:

```
hamilton(X,Y)←test(X,Y,empty).
test(X,Y,Ord)←ordering(Ord),first(Ord,X),last(Ord,Y),subset(succ(Ord),e).
test(X,Y,Ord)←test(X,Y,add(Ord,Z1,Z2)).
```

Notice that in the last rule above, we add a new pair $(\mathtt{Z}_1,\mathtt{Z}_2)$ to the Ord relation; the definition of add is quite simple and will be given later in the section. The variables $\mathtt{Z}_1$ and $\mathtt{Z}_2$ are existential but of lower order than the relation Ord. In other words, our transformation decreases by one the order of the existential predicate variables that the program contains. Thus, if we repeat this process successively, at the end we get a program that only contains existential variables of type ι. In the Hamilton program, one step suffices to complete the transformation (the variables $\mathtt{Z}_1$ and $\mathtt{Z}_2$ are of type ι).

We can now provide a more general description of the aforementioned transformation. Let P be a program and assume it contains the following rule:

$\mathtt{p}(\bar{\mathtt{X}}) \leftarrow \mathsf{B}[\bar{\mathtt{X}},\mathtt{R}].$

where $\mathsf{B}[\bar{\mathtt{X}},\mathtt{R}]$ is an expression containing the variables $\bar{\mathtt{X}} = \mathtt{X}_1,\ldots,\mathtt{X}_m$ that occur in the head of the rule and also a predicate variable $\mathtt{R}$ of order $k \geq 1$ that does not occur in the head and thus is existentially quantified. Let $\rho_{\mathtt{R}}$ be the type of $\mathtt{R}$. We replace the aforementioned rule with the following set of rules:

$\mathtt{p}(\bar{\mathtt{X}}) \leftarrow \mathtt{test}_{\mathsf{B}}(\bar{\mathtt{X}},\mathtt{empty}_{\rho_{\mathtt{R}}}).$
$\mathtt{test}_{\mathsf{B}}(\bar{\mathtt{X}},\mathtt{R}) \leftarrow \mathsf{B}[\bar{\mathtt{X}},\mathtt{R}].$
$\mathtt{test}_{\mathsf{B}}(\bar{\mathtt{X}},\mathtt{R}) \leftarrow \mathtt{test}_{\mathsf{B}}(\bar{\mathtt{X}},\mathtt{add}_{\rho_{\mathtt{R}}}(\mathtt{R},\bar{\mathtt{Z}})).$

The predicate $\texttt{empty}_{\rho_{\mathtt{R}}}$ defines the empty relation of objects of type $\rho_{\mathtt{R}}$ and the predicate $\texttt{add}_{\rho_{\mathtt{R}}}$ adds an element to a relation of type $\rho_{\mathtt{R}}$. The predicate $\texttt{eq}_{\rho_{\bar{\mathtt{Z}}}}$ is a higher-order equality predicate.

$\texttt{empty}_{\rho_{\mathtt{R}}}(\bar{\mathtt{Y}}) \leftarrow \texttt{false.}$
$\texttt{add}_{\rho_{\mathtt{R}}}(\mathtt{R},\bar{\mathtt{Z}},\bar{\mathtt{Y}}) \leftarrow \mathtt{R}(\bar{\mathtt{Y}}).$
$\texttt{add}_{\rho_{\mathtt{R}}}(\mathtt{R},\bar{\mathtt{Z}},\bar{\mathtt{Y}}) \leftarrow \texttt{eq}_{\rho_{\bar{\mathtt{Z}}}}(\bar{\mathtt{Z}},\bar{\mathtt{Y}}).$

Note that $\text{add}_{\rho_{\mathtt{R}}}(\mathrm{R},\bar{\mathrm{Z}})$ denotes a relation that contains every element $\bar{\mathtt{Y}}$ of $\mathtt{R}$ and also $\bar{\mathtt{Z}}$. The process introduces only the existential variables $\bar{\mathtt{Z}}$, which are of order at most $k-1$.

We have the following theorem, whose proof is given in the supplementary material.

Theorem 5.
Let P be a Higher-Order *Datalog*$^{\neg}$ program that defines a query $\mathcal{Q}$ under the well-founded semantics (resp. stable model semantics with cautious reasoning, stable model semantics with brave reasoning). Let P' be the program that results by applying the aforementioned transformation to some rule of P. Then, P' defines the same query $\mathcal{Q}$ under the well-founded semantics (resp. stable model semantics with cautious reasoning, stable model semantics with brave reasoning).

By applying the transformation described above to each rule and every existential variable of order k, we obtain a program without such variables. Repeating this process iteratively for variables of order $k-1$, $k-2$, and so on, we can eventually eliminate all existential predicate variables from the initial program.

If we denote with Higher-Order Datalog$^{\neg,\nexists}$ the fragment of Higher-Order Datalog$^{\neg}$ that does not contain existential predicate variables, then the following corollary is immediate:

Corollary 5.1.
$(k+1)$-Order Datalog$^{\neg,\nexists}$ and Stratified $(k+1)$-Order Datalog$^{\neg,\nexists}$ capture $k-\mathsf{EXP}$ under the well-founded semantics. $(k+1)$-Order Datalog$^{\neg,\nexists}$ and Stratified+Choices $(k+1)$-Order Datalog$^{\neg,\nexists}$ under the stable model semantics capture $\mathsf{co}-(k-\mathsf{NEXP})$ with cautious reasoning and $k-\mathsf{NEXP}$ with brave reasoning.

Obviously, Higher-Order Datalog$^{\neg,\nexists}$ under any of the aforementioned semantics captures $\mathsf{ELEMENTARY}$.

6 Conclusions and future work

We have presented an exploration of the expressive power of Higher-Order *Datalog*$^{\neg}$ under the well-founded and the stable model semantics. Our results identify fragments of the language that, despite being syntactically restricted, possess the same expressive power as the full language. Moreover, our results indicate that by increasing the order of programs under the well-founded semantics we can surpass the expressive power of lower-order programs under the stable model semantics.

There are several challenging directions for future work. First, although our results indicate that by increasing the order of the programs the well-founded semantics can match the power of stable model semantics, it is still unclear to us if there exists a formal transformation from a k-order program P to a $(k+1)$-order program P' such that the well-founded semantics of P' captures, in some sense, the stable model semantics of P. Another direction that would be very interesting and certainly quite challenging, would be the implementation of Higher-Order $Datalog^{\neg}$. In general, implementing efficiently non-monotonic extensions of Datalog is already non-trivial even at the first-order case. Probably, a promising direction would be to identify interesting subclasses of Higher-Order $Datalog^{\neg}$ that lend themselves to efficient implementation while at the same time retaining some strong expressibility features of the language. For example, as observed by one of the reviewers, it would be interesting to investigate the notion of *safety* in Higher-Order $Datalog^{\neg}$ and whether this notion affects the expressiveness and the potential for efficient implementation of the language.

Supplementary material

The supplementary material for this article can be found at http://dx.doi.org/10.1017/S1471068425100227.

Acknowledgments

This work was supported by a research project which is implemented in the framework of H.F.R.I call "Basic research Financing (Horizontal support of all Sciences)" under the National Recovery and Resilience Plan "Greece 2.0" funded by the European Union - NextGenerationEU (H.F.R.I. Project Number: 16116).

References

Amendola, G., Ricca, F. and Truszczynski, M. 2019. Beyond NP: Quantifying over answer sets. *Theory and Practice of Logic Programming* 19, 5-6, 705–721.

Bogaerts, B., Charalambidis, A., Chatziagapis, G., Kostopoulos, B., Pollaci, S. and Rondogiannis, P. 2024. The stable model semantics for higher-order logic programming. *Theory and Practice of Logic Programming* 24, 4, 737–754.

Bogaerts, B., Janhunen, T. and Tasharrofi, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *Theory and Practice of Logic Programming* 16, 5-6, 570–586.

Charalambidis, A., Nomikos, C. and Rondogiannis, P. 2019. The expressive power of higher-order datalog. *Theory and Practice of Logic Programming* 19, 5-6, 925–940.

Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys* 33, 3, 374–425.

Denecker, M., Marek, V. and TruszczyŃski, M. 2000. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Vol. 597 of The Springer International Series in Engineering and Computer Science, Springer US, 127–144.

Denecker, M., Marek, V. W. and Truszczynski, M. 2004. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Information and Computation* 192, 1, 84–121.

GrÄdel, E. 1992. Capturing complexity classes by fragments of second-order logic. *Theoretical Computer Science* 101, 1, 35–57.

Immerman, N. 1986. Relational queries computable in polynomial time. *Information and Control* 68, 1-3, 86–104.

Jones, N. D. 2001. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming* 11, 1, 5–94.

Kolaitis, P. G. 1991. The expressive power of stratified programs. *Information and Computation* 90, 1, 50–66.

Leivant, D. 1989. Descriptive characterizations of computational complexity. *Journal of Computer and System Science* 39, 1, 51–83.

Lichtenstein, D. and Sipser, M. 1980. GO is polynomial-space hard. *Journal of the ACM* 27, 2, 393–401.

NiemelÄ, I. 2008. Answer set programming without unstratified negation. In Logic Programming 2008, M. Garcia de la Banda and E. Pontelli, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 88–92.

Papadimitriou, C. H. 1985. A note on the expressive power of prolog. *Bulletin of the EATCS* 26, 21–22.

Vardi, M. Y. 1982. The complexity of relational query languages (extended abstract). In Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5–7, 1982, ACM, San Francisco, CA, USA, 137–146.