

# *The Expressive Power of Higher-Order Datalog*

ANGELOS CHARALAMBIDIS

*Institute of Informatics and Telecommunications, NCSR “Demokritos”, Greece  
(e-mail: [acharal@iit.demokritos.gr](mailto:acharal@iit.demokritos.gr))*

CHRISTOS NOMIKOS

*Dept of Computer Science and Engineering, University of Ioannina, Greece  
(e-mail: [cnomikos@cs.uoi.gr](mailto:cnomikos@cs.uoi.gr))*

PANOS RONDOGIANNIS

*Dept of Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece  
(e-mail: [prondo@di.uoa.gr](mailto:prondo@di.uoa.gr))*

*submitted 29 July 2019; accepted 31 July 2019*

---

## Abstract

A classical result in descriptive complexity theory states that Datalog expresses exactly the class of polynomially computable queries on ordered databases (Papadimitriou 1985; Grädel 1992; Vardi 1982; Immerman 1986; Leivant 1989). In this paper we extend this result to the case of higher-order Datalog. In particular, we demonstrate that on ordered databases, for all  $k \geq 2$ ,  $k$ -order Datalog captures  $(k - 1)$ -EXPTIME. This result suggests that higher-order extensions of Datalog possess superior expressive power and they are worthwhile of further investigation both in theory and in practice.

**KEYWORDS:** Datalog, Higher-Order Logic Programming, Descriptive Complexity Theory.

---

## 1 Introduction

Higher-order programming languages are widely recognized as offering a more modular and expressive way of programming. The use of higher-order constructs in functional programming has been a key factor for the development and success of the functional paradigm. Functional programmers have embraced the higher-order style of programming because they have realized that it offers significant advantages in everyday programming. Apart from the empirical evidence of the strengths of higher-order functions, there also exist concrete theoretical results that support this claim. For example, it has been demonstrated (Jones 2001) that if we restrict attention to a functional language that is not Turing-complete, then its higher-order fragments capture broader complexity classes than the lower-order ones.

The situation in logic programming is not so clear-cut. Logic programming languages have traditionally been first-order, offering to programmers only certain restricted higher-order capabilities. There have been some serious attempts to develop general-purpose higher-order logic programming languages, most notably Hilog (Chen et al. 1993) and  $\lambda$ -Prolog (Miller and Nadathur 1986). Although these systems have not become mainstream, they have found some remarkable application domains beyond those of traditional logic programming. For example,  $\lambda$ -Prolog has been used for theorem-proving

and program analysis. Moreover, Hilog's ideas have been incorporated in the Flora-2 system which has been used for meta-programming (Yang et al. 2003) and data integration (Lovrenčić and Čubriilo 1999). Also, it has recently been demonstrated that higher-order logic programming can be used to concisely represent complicated user-preferences in deductive databases (Charalambidis et al. 2018), with some demonstrated applications in airline reservation and movie-selection systems. All the above applications suggest that higher-order logic programming can open new, fresh, and promising directions for logic programming as a whole.

In this paper we provide theoretical results that affirm the power of higher-order logic programming. Intuitively speaking, we demonstrate that Higher-Order Datalog possesses superior expressive power compared to classical Datalog. Our results belong to a research stream that studies the expressive power of fragments or extensions of logic programming languages using complexity-theoretic tools. A classical expressibility theorem in this area states that on ordered databases Datalog captures PTIME (Papadimitriou 1985; Grädel 1992; Vardi 1982; Immerman 1986; Leivant 1989). This is an interesting (and somewhat unexpected) result, because it suggests that a seemingly simple language can express all polynomially computable queries. In this paper we extend this classical result to the case of Higher-Order Datalog. More specifically:

- We demonstrate that every language decided by a  $k$ -order Datalog program,  $k \geq 2$ , can also be decided by a  $(k - 1)$ -exponential-time bounded Turing machine. This result relies on developing a bottom-up proof procedure for  $k$ -order Datalog programs, which generalizes the familiar one for classical Datalog programs.
- We demonstrate that every language decided by a  $(k - 1)$ -exponential-time bounded Turing machine,  $k \geq 2$ , can also be decided by a  $k$ -order Datalog program. The proof actually involves a simulation of the Turing machine by the Higher-Order Datalog program. Our simulation uses the encoding of “big numbers” by higher-order functions (relations in our case) used in (Jones 2001).

The above results essentially demonstrate that higher-order extensions of Datalog possess superior expressive power than classical Datalog and they are worthwhile of further investigation both in theory and in practice. Additionally, the results show a striking analogy with the expressibility results of (Jones 2001) regarding higher-order “read-only” functional programs. It may be possible that this analogy can be further exploited to derive additional complexity-theoretic insights for interesting classes of Higher-Order Datalog programs. This possibility is discussed in the concluding section of the paper.

The rest of the paper is organized as follows. Section 2 introduces the syntax and the semantics of Higher-Order Datalog. Section 3 presents the underlying framework for connecting logic programming with complexity theory. Section 4 demonstrates that on ordered databases, for all  $k \geq 2$ ,  $k$ -order Datalog captures  $(k - 1)$ -EXPTIME. Section 5 gives pointers to future work. Detailed proofs are included in the supplementary material corresponding to this paper at the TPLP archives. In particular it contains a version of the proof that Datalog captures PTIME on ordered databases; this is needed because our developments are based on extending this result. Moreover, it contains the one direction of our proof that  $k$ -order Datalog captures  $(k - 1)$ -EXPTIME.

## 2 Higher-Order Datalog

### 2.1 The Syntax of Higher-Order Datalog

The language *Higher-Order Datalog* that we consider in this paper is the function-free subset of the higher-order logic programming language  $\mathcal{H}$  introduced in (Charalambidis et al. 2013). Higher-Order Datalog inherited from  $\mathcal{H}$  an important syntactic restriction which ensures that the language retains the usual least fixpoint semantics of classical logic programming. This syntactic restriction was proposed many years ago by W. W. Wadge (Wadge 1991) (and also later independently by M. Bezem (Bezem 1999)):

**The higher-order syntactic restriction.** *In the head of every rule in a program, each argument of predicate type must be a variable, and all such variables must be distinct.*

#### Example 1

The following is a legitimate higher-order program that defines the union of two relations  $P, Q$  (for the moment we use ad-hoc Prolog-like syntax):

```
union(P,Q,X):-P(X).
union(P,Q,X):-Q(X).
```

However, the following program does not satisfy Wadge's restriction:

```
q(a).
r(q).
```

because the predicate constant  $q$  appears as an argument in the head of a rule. Similarly, the program:

```
p(Q,Q):-Q(a).
```

is problematic because the predicate variable  $Q$  is used twice in the head of the rule.  $\square$

We now proceed to the exact definition of the syntax of Higher-Order Datalog. The language is based on a simple type system that supports two base types:  $o$ , the boolean domain, and  $\iota$ , the domain of individuals (data objects). The composite types are partitioned into two classes: predicate (assigned to predicate symbols) and argument (assigned to parameters of predicates).

#### Definition 1

*Predicate* and *argument* types, denoted by  $\pi$  and  $\rho$  respectively, are defined as follows:

$$\begin{aligned}\pi &:= o \mid (\rho \rightarrow \pi) \\ \rho &:= \iota \mid (\rho \rightarrow \pi)\end{aligned}$$

As usual, the binary operator  $\rightarrow$  is right-associative. It can be easily seen that every predicate type  $\pi$  can be written in the form  $\rho_1 \rightarrow \dots \rightarrow \rho_n \rightarrow o$ ,  $n \geq 0$ , and  $n$  will be called the *arity* of the type  $\pi$ ; for  $n = 0$  we assume that  $\pi = o$ . We proceed by defining the syntax of Higher-Order Datalog:

#### Definition 2

The *alphabet* of Higher-Order Datalog consists of the following:

- Predicate constants of every predicate type  $\pi$  (denoted by lowercase letters or words that start with lowercase letters such as  $p, q, \text{is\_zero}, \dots$ ).

- Predicate variables of every predicate type  $\pi$  (denoted by capital letters such as  $P, Q, R, \dots$ ).
- Individual constants of type  $\iota$  (denoted by lowercase letters or words that start with lowercase letters such as  $a, b, \text{end}, \dots$ ).
- Individual variables of type  $\iota$  (denoted by capital letters such as  $X, Y, Z, \dots$ ).
- The inverse implication constant  $\leftarrow$ , the conjunction symbol  $\wedge$ , the left and right parentheses, and the equality constant  $\approx$  for comparing terms of type  $\iota$ .

It will always be obvious from context whether a variable name that we use is a predicate or individual one; similarly for the names of predicate and individual constants.

Predicate constants correspond to the predicates that are defined in a program, while predicate and individual variables are used as formal parameters in such predicate definitions. The set consisting of the predicate variables and the individual variables will be called the set of *argument variables*. Argument variables will be usually denoted by  $V$  and its subscripted versions.

#### Definition 3

The set of *terms* of Higher-Order Datalog is defined as follows:

- Every predicate variable (respectively predicate constant) of type  $\pi$  is a term of type  $\pi$ ; every individual variable (respectively individual constant) of type  $\iota$  is a term of type  $\iota$ ;
- if  $E_1$  is a term of type  $\rho \rightarrow \pi$  and  $E_2$  a term of type  $\rho$  then  $(E_1 E_2)$  is a term of type  $\pi$ .

#### Definition 4

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

- A term of type  $\rho$  is an expression of type  $\rho$ ;
- if  $E_1$  and  $E_2$  are terms of type  $\iota$ , then  $(E_1 \approx E_2)$  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  $o$  will often be referred to as *atoms*. We will omit parentheses when no confusion arises. To denote that an expression  $E$  has type  $\rho$  we will often write  $E : \rho$ .

#### Definition 5

A *clause* (or *rule*) of Higher-Order Datalog is a formula  $p V_1 \dots V_n \leftarrow E_1 \wedge \dots \wedge E_m$ , where  $p$  is a predicate constant of type  $\rho_1 \rightarrow \dots \rightarrow \rho_n \rightarrow o$ ,  $V_1, \dots, V_n$ ,  $n \geq 0$ , are argument variables of types  $\rho_1, \dots, \rho_n$  respectively, and  $E_1, \dots, E_m$ ,  $m \geq 0$ , are atoms. The term  $p V_1 \dots V_n$  is called the *head* of the clause, the variables  $V_1, \dots, V_n$  are the *formal parameters* of the clause and the conjunction  $E_1 \wedge \dots \wedge E_m$  is its *body*. A *definitional clause* is a clause that additionally satisfies the following two restrictions:

1. All the formal parameters are distinct variables (i.e., for all  $i, j$  such that  $1 \leq i, j \leq n$  and  $i \neq j$ ,  $V_i \neq V_j$ ).
2. The only variables that can appear in the body of the clause are its formal parameters and possibly some additional individual variables (variables of type  $\iota$ ).

A *definitional program*  $P$  of Higher-Order Datalog is a set of definitional program clauses.

In the rest of the paper, when we refer to “clauses” and “programs” we will mean definitional ones. Notice that for uniformity reasons, the above definition requires that *all* formal parameters are distinct, even the type  $\iota$  ones. This is not a real restriction, because two occurrences of the same individual variable in the head of a clause can be replaced by distinct variables, which are then explicitly equated in the body of the clause using the constant  $\approx$ .

*Example 2*

Assume that  $\mathbf{p}$  is of type  $\iota \rightarrow o$ ,  $\mathbf{q}$  of type  $\iota \rightarrow \iota \rightarrow o$  and  $\mathbf{r}$  of type  $(\iota \rightarrow o) \rightarrow (\iota \rightarrow o) \rightarrow \iota \rightarrow o$ . The following is a legitimate program of Higher-Order Datalog:

$$\begin{aligned} \mathbf{p} \ X &\leftarrow (X \approx \mathbf{a}) \\ \mathbf{q} \ X \ Y &\leftarrow (X \approx Y) \\ \mathbf{r} \ P \ Q \ X &\leftarrow (X \approx \mathbf{b}) \wedge (P \ X) \wedge (Q \ Y) \end{aligned}$$

Notice that the formal parameters of every clause are distinct.

The above somewhat rigid syntax is quite convenient for formal purposes. We will use it in the following subsection when discussing the semantics of Higher-Order Datalog. However, in the rest of the paper we will relax the above strict notation and write in a more Prolog-like syntax.

*Example 3*

The program of the previous example will be written in the following simpler form:

$$\begin{aligned} \mathbf{p} \ \mathbf{a}. \\ \mathbf{q} \ X \ X. \\ \mathbf{r} \ P \ Q \ \mathbf{b} &\leftarrow (P \ \mathbf{b}), (Q \ Y). \end{aligned}$$

In other words, we will allow some common Prolog conventions, such as the usual fact syntax, using the comma instead of  $\wedge$ , allowing individual constants to appear in the heads of clauses, having multiple occurrences of the same individual variable in the head of a clause, and using the full stop to end clauses. Obviously, every program that uses the Prolog-like syntax can be transformed into the more formal one.

*Remark*

The syntax described above, although somewhat close to the traditional syntax of first-order logic programming, differs from it in one important respect: when defining predicates, we do not use the common tuple notation but instead we have adopted the “*currying syntax*” that is standard in functional programming languages. For example, in the head of the clause defining the predicate  $\mathbf{r}$  above, we write  $\mathbf{r} \ P \ Q \ \mathbf{b}$  instead of the more common  $\mathbf{r}(P, Q, \mathbf{b})$ . Currying is an important tool that allows functions (or predicates in our case) to be *partially applied*, ie., invoked with less arguments than their full arity. As we are going to see in the next sections, partial applications play an important role in our constructions.

The *Herbrand universe*  $U_P$  of a program  $P$  is the set of constants that appear in  $P$  (if no constant appears in  $P$ , we select an arbitrary one).

In the rest of the paper we will consider fragments of Higher-Order Datalog based on the *order* of predicates that appear in programs:

*Definition 6*

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

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

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

*Definition 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  $k - 1$ .

*Example 4*

Consider again the program of Example 2. Predicates **p** and **q** are ordinary first-order ones. The order of **r** is:

$$\text{order}((\iota \rightarrow o) \rightarrow (\iota \rightarrow o) \rightarrow \iota \rightarrow o) = 1 + \max(\{\text{order}(\iota \rightarrow o), \text{order}(\iota)\}) = 2$$

This program belongs to 2nd-order Datalog because the predicate constants **p**, **q** and **r** have order less than or equal to 2 and the predicate variables **P** and **Q** have order 1.

## 2.2 The Semantics of Higher-Order Datalog

In this subsection we present the semantics of Higher-Order Datalog, which is based on the ideas initially proposed in (Wadge 1991) and subsequently extended and refined in (Kountouriotis et al. 2005; Charalambidis et al. 2013). The key idea is to interpret program predicates as monotonic relations. This ensures that the immediate consequence operator of the program (see Definition 13 later in this subsection), is also monotonic and therefore has a least fixpoint.

We start by defining the semantics of the types of our language. More specifically, we define simultaneously and recursively the semantics  $\llbracket \rho \rrbracket$  of a type  $\rho$  and a corresponding partial order  $\sqsubseteq_\rho$  on the elements of  $\llbracket \rho \rrbracket$ . We adopt the usual ordering of the truth values *false* and *true*, i.e. *false*  $\leq$  *false*, *true*  $\leq$  *true* and *false*  $\leq$  *true*. Given posets  $A$  and  $B$ , we write  $[A \xrightarrow{m} B]$  to denote the set of all monotonic functions from  $A$  to  $B$ .

*Definition 8*

Let **P** be a program. Then:

- $\llbracket \iota \rrbracket = U_P$  and  $\sqsubseteq_\iota$  is the trivial partial order that relates every element of  $U_P$  to itself;
- $\llbracket o \rrbracket = \{\text{false}, \text{true}\}$  and  $\sqsubseteq_o$  is the partial order  $\leq$  on truth values;
- $\llbracket \rho \rightarrow \pi \rrbracket = [\llbracket \rho \rrbracket \xrightarrow{m} \llbracket \pi \rrbracket]$  and  $\sqsubseteq_{\rho \rightarrow \pi}$  is the partial order defined as follows: for all  $f, g \in \llbracket \rho \rightarrow \pi \rrbracket$ ,  $f \sqsubseteq_{\rho \rightarrow \pi} g$  iff  $f(d) \sqsubseteq_\pi g(d)$  for all  $d \in \llbracket \rho \rrbracket$ .

We now proceed to define Herbrand interpretations and states.

*Definition 9*

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

- to each individual constant  $c$  that appears in  $P$ , the element  $I(c) = c$ ;
- to each predicate constant  $p : \pi$  that appears in  $P$ , an element  $I(p) \in \llbracket \pi \rrbracket$ ;

*Definition 10*

A *Herbrand state*  $s$  of a program  $P$  is a function that assigns to each argument variable  $V$  of type  $\rho$ , an element  $s(V) \in \llbracket \rho \rrbracket$ .

In the following,  $s[V_1/d_1, \dots, V_n/d_n]$  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 11*

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:

- $\llbracket V \rrbracket_s(I) = s(V)$ ;
- $\llbracket c \rrbracket_s(I) = I(c)$ ;
- $\llbracket p \rrbracket_s(I) = I(p)$ ;
- $\llbracket (E_1 \ E_2) \rrbracket_s(I) = \llbracket E_1 \rrbracket_s(I)(\llbracket E_2 \rrbracket_s(I))$ ;
- $\llbracket (E_1 \approx E_2) \rrbracket_s(I) = \text{true}$  if  $\llbracket E_1 \rrbracket_s(I) = \llbracket E_2 \rrbracket_s(I)$  and *false* otherwise.

For ground expressions  $E$  we will often write  $\llbracket E \rrbracket(I)$  instead of  $\llbracket E \rrbracket_s(I)$  since in this case the meaning of  $E$  is independent of  $s$ . The notion of *model* is defined as follows:

*Definition 12*

Let  $P$  be a program and  $M$  be a Herbrand interpretation of  $P$ . Then,  $M$  is a *Herbrand model* of  $P$  iff for every clause  $p \ V_1 \cdots V_n \leftarrow E_1 \wedge \cdots \wedge E_m$  in  $P$  and for every Herbrand state  $s$ , if for all  $i \in \{1, \dots, m\}$ ,  $\llbracket E_i \rrbracket_s(M) = \text{true}$  then  $\llbracket p \ V_1 \cdots V_n \rrbracket_s(M) = \text{true}$ .

*Example 5*

Consider the following program, where  $p$  is of type  $\iota \rightarrow o$  and  $q$  of type  $(\iota \rightarrow o) \rightarrow o$ :

$$\begin{array}{l} p \ a. \\ q \ R \leftarrow (R \ b). \end{array}$$

It can easily be seen that the Herbrand interpretation that assigns to  $p$  the relation  $\{a\}$  and to  $q$  the relation  $\{\{b\}, \{a, b\}\}$ , is a model of the program. Notice that the meaning of  $q$  is monotonic: since it is true of the relation  $\{b\}$ , it has to also be true of the relation  $\{a, b\}$  (which is a superset of  $\{b\}$ ). Actually, the interpretation we just described is the *minimum model* of the program, a notion that will be discussed shortly.

We denote the set of Herbrand interpretations of a program  $P$  with  $\mathcal{I}_P$ , and define a partial order on  $\mathcal{I}_P$  as follows: for all  $I, J \in \mathcal{I}_P$ ,  $I \sqsubseteq_{\mathcal{I}_P} J$  iff for every predicate constant  $p : \pi$  that appears in  $P$ ,  $I(p) \sqsubseteq_{\pi} J(p)$ . It is easy to prove that  $(\mathcal{I}_P, \sqsubseteq_{\mathcal{I}_P})$  is a complete lattice; we denote by  $\sqcup$  the least upper bound operation and by  $\perp_{\mathcal{I}_P}$  the least element of the lattice, with respect to  $\sqsubseteq_{\mathcal{I}_P}$ . Intuitively,  $\perp_{\mathcal{I}_P}$  assigns to every program predicate in  $P$  the empty relation.

We can now define the *immediate consequence operator* for Higher-Order Datalog programs, which generalizes the corresponding operator for classical Datalog (Lloyd 1987).

*Definition 13*

Let  $P$  be a program. The mapping  $T_P : \mathcal{I}_P \rightarrow \mathcal{I}_P$  is called the *immediate consequence operator* for  $P$  and is defined for every predicate constant  $p : \rho_1 \rightarrow \dots \rightarrow \rho_n \rightarrow o$  and  $d_i \in \llbracket \rho_i \rrbracket$  as:

$$T_P(I)(p) \ d_1 \dots d_n = \begin{cases} \text{true,} & \text{if there exists a clause } p \ V_1 \dots V_n \leftarrow E_1 \wedge \dots \wedge E_m \text{ in } P \text{ and} \\ & \text{a Herbrand state } s, \text{ such that } \llbracket E_i \rrbracket_{s[V_1/d_1, \dots, V_n/d_n]}(I) = \text{true} \\ & \text{for all } i \in \{1, \dots, m\} \\ \text{false,} & \text{otherwise.} \end{cases}$$

Define now the following sequence of interpretations:

$$\begin{aligned} T_P \uparrow 0 &= \perp_{\mathcal{I}_P} \\ T_P \uparrow (n+1) &= T_P(T_P \uparrow n) \\ T_P \uparrow \omega &= \bigsqcup \{T_P \uparrow n \mid n < \omega\} \end{aligned}$$

We then have the following theorem (see (Wadge 1991; Kountouriotis et al. 2005; Charalambidis et al. 2013) for more details):

*Theorem 1*

Let  $P$  be a program and let  $M_P = T_P \uparrow \omega$ . Then,  $M_P$  is the least Herbrand model of  $P$  and the least fixpoint of  $T_P$  (with respect to the ordering relation  $\sqsubseteq_{\mathcal{I}_P}$ ).

### 3 Decision Problems, Logic Programming, and Complexity Classes

In this section we initiate our investigation regarding the expressive power of Higher-Order Datalog. Our development is based on well-known ideas relating logic programming languages with complexity theory (see for example (Dantsin et al. 2001) for an introduction of the main concepts).

Let  $\Sigma$  be an alphabet. Without loss of generality, we fix  $\Sigma = \{a, b\}$ . Our goal is to demonstrate that sublanguages of Higher-Order Datalog correspond to interesting complexity classes over  $\Sigma$ . We first need to specify how strings over  $\Sigma$  can be encoded in our setting. For this purpose, we use a ternary predicate `input` which encodes in an *ordered* manner the input string. For example, to represent the string `abba` we use the four facts:

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

More generally, an input of length  $n > 0$  over  $\Sigma$  can be encoded with  $n$  facts of the above form. Moreover, for 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 referred as the *ordered database assumption*.



We will assume that every program defines a propositional **accept** predicate which, intuitively, signals whether a particular input string is accepted by our program. We have the following two definitions:

*Definition 14*

Let  $\Sigma$  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 minimum Herbrand model of  $P \cup \mathcal{D}_w$ .

*Definition 15*

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

Assuming the above representation of input strings through the **input** relation, the following classical result has been obtained in many different contexts (Papadimitriou 1985; Grädel 1992; Vardi 1982; Immerman 1986; Leivant 1989):

*Theorem 2*

The set of first-order Datalog programs captures PTIME.

We give a detailed proof of this theorem in the supplementary material corresponding to this paper at the TPLP archives by refining the expositions given in (Papadimitriou 1985) and (Dantsin et al. 2001). Although this is a well-known result, the reader is advised to first look through this proof before attempting to read the more involved ones in the rest of the paper. Actually, several ideas and predicates used in the first-order (see supplementary material) are extended to define the predicates for the higher-order case.

We now proceed to examine the expressive power of Higher-Order Datalog. We will need the following family of functions:

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

For all  $k \geq 0$ , the complexity class  $k$  – EXPTIME is defined as follows:

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

Notice that  $0$  – EXPTIME coincides with PTIME. The following theorem, which we will establish, is an extension of Theorem 2 to the case of Higher-Order Datalog:

*Theorem 3*

For every  $k \geq 1$ , the set of  $k$ -order Datalog programs captures  $(k - 1)$ -EXPTIME.

Obviously, for  $k = 1$  the above theorem gives as a special case Theorem 2. The detailed proof of Theorem 3 for the cases  $k \geq 2$ , is developed in the next section.

## 4 Higher-Order Datalog and Exponential Time Bounded Turing Machines

In order to establish Theorem 3, we prove two lemmas. The first one shows that every language decided by a  $k$ -order Datalog program can also be decided by a  $(k - 1)$ -exponential time bounded Turing machine. More specifically:

*Lemma 1*

Let  $P$  be a  $k$ -order Datalog program,  $k \geq 2$ , that decides a language  $L$ . Then, there exists a Turing machine that decides  $L$  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  $P$ .

The proof of the above lemma is given in the Appendix B of the supplementary material, and is based on calculating the time-complexity of the bottom-up proof procedure for Higher-Order Datalog.

We now demonstrate the following lemma, which is the converse of Lemma 1:

*Lemma 2*

Let  $M$  be a deterministic Turing machine that decides a language  $L$  in  $(k-1)$ -EXPTIME,  $k \geq 2$ . Then, there exists a  $k$ -order Datalog program  $P$  that decides  $L$ .

The next three subsections establish the proof of the above lemma. The key idea is to construct a  $k$ -order Datalog program that simulates the  $(k-1)$ -exponential-time-bounded Turing machine  $M$ . In order to achieve this, we must use the power of higher-order relations to represent “large numbers” that count the execution steps of the machine. As it turns out, by increasing the order of the programs that we use, we can increase the range of representable numbers.

Assume that  $M$  decides  $L$  in time  $O(\exp_{k-1}(n^q))$ . Then there exists an integer constant  $d$ , such that for every input  $w$  of length  $n \geq 2$ ,  $M$  terminates after at most  $\exp_{k-1}(n^d) - 1$  steps. The simulation that we will present, produces the correct answer for all inputs of length at least 2 by simulating  $\exp_{k-1}(n^d) - 1$  steps of the Turing machine  $M$ . Similarly to the first-order case (see the supplementary material), for the special cases of strings of length 0 or 1 that belong to  $L$ , the correct answer is produced directly by appropriate rules.

In Subsection 4.1 we demonstrate that for any  $d > 0$ , there exists a second-order Datalog program which, given any input relation of size  $n$ , can represent all natural numbers up to  $2^{n^d} - 1$ . In Subsection 4.2 we show that using  $k$ -order Datalog,  $k > 2$ , we can represent numbers up to  $\exp_{k-1}(n^d) - 1$ . Notice that the case  $k = 2$  has some differences from the case where  $k > 2$ , and that’s why we devote two different subsections to the two cases. The differences are due to the fact that the simulation for  $k = 2$  uses tuples in order to represent numbers, while the simulation for  $k > 2$  uses higher-order predicates for the same purpose.

Finally, in Subsection 4.3 we provide the actual simulation of the  $(k-1)$ -exponential-time-bounded Turing machine  $M$  by the  $k$ -order Datalog program.

### 4.1 The Second-Order Case

In this subsection we demonstrate that we can use second-order relations to represent numbers up to  $2^{n^d} - 1$ . We use a technique similar to the one introduced in (Jones 2001): a number in the range  $0, \dots, 2^{n^d} - 1$  can be represented by a function  $f : \{0, \dots, n^d - 1\} \rightarrow \{0, 1\}$ . Such a function is equivalent to a string of  $n^d$  binary digits. We assume that  $f(0)$  is the rightmost bit of the number and  $f(n^d - 1)$  the leftmost one. Such a string can represent any number in the required range.

Second-order Datalog can implement a function such as the above using a binary predicate  $p \bar{X} \forall$ , where  $\bar{X} = X_1 \cdots X_d$  is a  $d$ -tuple (alternatively,  $d$  consecutive arguments)

that can represent all numbers in the range 0 to  $n^d - 1$  (please see Appendix A of supplementary material), and  $V$  is a variable that can receive either the constant **low** or the constant **high** (corresponding to 0 and 1 respectively).

The main predicates that are defined in this subsection are: **zero**<sub>1</sub>, **last**<sub>1</sub>, **is\_zero**<sub>1</sub>, **is\_non\_zero**<sub>1</sub>, **is\_last**<sub>1</sub>, **non\_last**<sub>1</sub>, **pred**<sub>1</sub>, **succ**<sub>1</sub>, **equal**<sub>1</sub>, and **less\_than**<sub>1</sub>. We explain the purpose of each one of them, just before we define it. The subscript 1 in all the above predicates, denotes that we are now using first-order relations in order to represent our numbers (in the first order we represented smaller numbers using  $d$ -tuples). Notice that we will also use some additional auxiliary predicates in our definitions as-well-as some predicates from the first-order case defined in the supplementary material (namely, **tuple\_zero**, **tuple\_last**, **tuple\_pred**).

We start by defining the predicates **zero**<sub>1</sub> and **last**<sub>1</sub> that represent the first and last numbers of the range (namely 0 and  $2^{n^d} - 1$  respectively).

$$\begin{aligned}\text{zero}_1 \bar{X} &\text{ low.} \\ \text{last}_1 \bar{X} &\text{ high.}\end{aligned}$$

Notice that **zero**<sub>1</sub> returns in its second argument the value **low** for all values of its first argument (and similarly for **last**<sub>1</sub> and **high**). We now define a predicate **is\_zero**<sub>1</sub> that checks if its argument is equal to the function **zero**<sub>1</sub>. The auxiliary predicate (**all\_to\_right**<sub>1</sub>  $V N \bar{X}$ ) checks if all the bits of  $N$  starting from the position indicated by  $\bar{X}$  until the right end of  $N$ , have the value  $V$  (**low** in our case).

$$\begin{aligned}\text{is\_zero}_1 N &\leftarrow (\text{tuple\_last } \bar{X}), (\text{all\_to\_right}_1 \text{ low } N \bar{X}). \\ \text{all\_to\_right}_1 V N \bar{X} &\leftarrow (\text{tuple\_zero } \bar{X}), (N \bar{X} V). \\ \text{all\_to\_right}_1 V N \bar{X} &\leftarrow (\text{tuple\_pred } \bar{X} \bar{Y}), (N \bar{X} V), (\text{all\_to\_right}_1 V N \bar{Y}).\end{aligned}$$

Similarly we define the predicate **is\_non\_zero**<sub>1</sub> that succeeds if its argument is not equal to the function **zero**<sub>1</sub>. The predicate (**exists\_to\_right**<sub>1</sub>  $V N \bar{X}$ ) checks if there exists a bit of  $N$ , starting from the position indicated by  $\bar{X}$  until the right end of  $N$ , that has the value  $V$  (**high** in our case).

$$\begin{aligned}\text{non\_zero}_1 N &\leftarrow (\text{tuple\_last } \bar{X}), (\text{exists\_to\_right}_1 \text{ high } N \bar{X}). \\ \text{exists\_to\_right}_1 V N \bar{X} &\leftarrow (N \bar{X} V). \\ \text{exists\_to\_right}_1 V N \bar{X} &\leftarrow (\text{tuple\_pred } \bar{X} \bar{Y}), (\text{exists\_to\_right}_1 V N \bar{Y}).\end{aligned}$$

Symmetrically, we can define the predicates **is\_last**<sub>1</sub> and **non\_last**<sub>1</sub>, as follows:

$$\begin{aligned}\text{is\_last}_1 N &\leftarrow (\text{tuple\_last } \bar{X}), (\text{all\_to\_right}_1 \text{ high } N \bar{X}). \\ \text{non\_last}_1 N &\leftarrow (\text{tuple\_last } \bar{X}), (\text{exists\_to\_right}_1 \text{ low } N \bar{X}).\end{aligned}$$

Next we define **pred**<sub>1</sub> to capture the notion of the predecessor of a number. This is one of the cases where partial application and currying (see Remark in Section 2.1) plays an important role in our encoding of big numbers. More specifically, the predicate **pred**<sub>1</sub> is different than the **tuple\_pred** predicate, in the sense that it does not check if one number is the predecessor of another number; instead, if  $N$  is the representation of a number  $n$  then the partially applied expression (**pred**<sub>1</sub>  $N$ ) is the representation of the predecessor of  $N$ . The idea is that the predecessor of  $N$ , namely (**pred**<sub>1</sub>  $N$ ), is a number whose binary representation has at position  $\bar{X}$  either: (i) the same binary digit as  $N$  if there exists some bit of  $N$  that is on the right of position  $\bar{X}$  that has the value **high**, or (ii) the inverse

binary digit of that of  $N$  at position  $\bar{X}$ , if all the bits of  $N$  that are on the right of  $\bar{X}$  have the value low.

$$\begin{aligned} \text{pred}_1 N \bar{X} V &\leftarrow (\text{tuple\_zero } \bar{X}), (\text{non\_zero}_1 N), \\ &\quad (N \bar{X} V1), (\text{invert } V1 V). \\ \text{pred}_1 N \bar{X} V &\leftarrow (\text{non\_zero}_1 N), (\text{tuple\_pred } \bar{X} \bar{Y}), \\ &\quad (\text{exists\_to\_right}_1 \text{ high } N \bar{Y}), (N \bar{X} V). \\ \text{pred}_1 N \bar{X} V &\leftarrow (\text{non\_zero}_1 N), (\text{tuple\_pred } \bar{X} \bar{Y}), \\ &\quad (\text{all\_to\_right}_1 \text{ low } N \bar{Y}), \\ &\quad (N \bar{X} V1), (\text{invert } V1 V). \\ \text{invert low high.} \\ \text{invert high low.} \end{aligned}$$

Symmetrically, way define  $\text{succ}_1$  which gives the successor of a given number:

$$\begin{aligned} \text{succ}_1 N \bar{X} V &\leftarrow (\text{tuple\_zero } \bar{X}), (\text{non\_last}_1 N), \\ &\quad (N \bar{X} V1), (\text{invert } V1 V). \\ \text{succ}_1 N \bar{X} V &\leftarrow (\text{non\_last}_1 N), (\text{tuple\_pred } \bar{X} \bar{Y}), \\ &\quad (\text{exists\_to\_right}_1 \text{ low } N \bar{Y}), (N \bar{X} V). \\ \text{succ}_1 N \bar{X} V &\leftarrow (\text{non\_last}_1 N), (\text{tuple\_pred } \bar{X} \bar{Y}), \\ &\quad (\text{all\_to\_right}_1 \text{ high } N \bar{Y}), \\ &\quad (N \bar{X} V1), (\text{invert } V1 V). \end{aligned}$$

We will also need the equality of two numbers  $N$  and  $M$ . We compare them bit by bit, starting from the leftmost possible position and moving to the left.

$$\begin{aligned} \text{equal}_1 N M &\leftarrow (\text{tuple\_last } \bar{X}), (\text{equal\_test}_1 N M \bar{X}). \\ \text{equal\_test}_1 N M \bar{X} &\leftarrow (\text{tuple\_zero } \bar{X}), (N \bar{X} V), (M \bar{X} V). \\ \text{equal\_test}_1 N M \bar{X} &\leftarrow (\text{tuple\_pred } \bar{X} \bar{Y}), (N \bar{X} V), (M \bar{X} V), \\ &\quad (\text{equal\_test}_1 N M \bar{Y}). \end{aligned}$$

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

$$\begin{aligned} \text{less\_than}_1 N M &\leftarrow (\text{is\_zero}_1 N), (\text{non\_zero}_1 M). \\ \text{less\_than}_1 N M &\leftarrow (\text{non\_zero}_1 N), (\text{non\_zero}_1 M), \\ &\quad (\text{less\_than}_1 (\text{pred}_1 N) (\text{pred}_1 M)). \end{aligned}$$

In order to represent even larger numbers, we need to extend the above predicates to higher orders. This requires certain modifications to the predicate definitions, as the following subsection demonstrates.

## 4.2 Extending to Arbitrary Orders

To define numbers larger than  $2^{n^d}$ , we need to generalize the ideas of the previous subsection. For example, a number in the range  $0, \dots, 2^{2^{n^d}} - 1$  can be represented by a function  $f : \{0, \dots, 2^{2^{n^d}} - 1\} \rightarrow \{0, 1\}$ . In other words, to define the numbers and the operations at level  $k + 1$ , for  $k \geq 2$ , we need to use the numbers and the operations of level  $k$ . The definitions we give below, have certain differences from the second-order ones given in the previous subsection. This is due to the fact that the second-order predicates, use the tuple-based predicates that are defined in (the Appendix A of the supplementary material) (while the ones we define below, do not). We start with  $\text{zero}_{k+1}$  and  $\text{last}_{k+1}$ .

Notice that the parameter  $X$  is now a relation (and not a tuple as in the case of  $\text{zero}_1$  and  $\text{last}_1$ ).

$\text{zero}_{k+1} \ X \ \text{low}.$   
 $\text{last}_{k+1} \ X \ \text{high}.$

We now define  $\text{is\_zero}_{k+1}$  which succeeds if its argument is equal to  $\text{zero}_{k+1}$ :

$\text{is\_zero}_{k+1} \ N \quad \leftarrow \quad (\text{all\_to\_right}_{k+1} \ \text{low} \ N \ \text{last}_k).$   
 $\text{all\_to\_right}_{k+1} \ V \ N \ X \quad \leftarrow \quad (\text{is\_zero}_k \ X), (N \ X \ V).$   
 $\text{all\_to\_right}_{k+1} \ V \ N \ X \quad \leftarrow \quad (\text{non\_zero}_k \ X), (N \ X \ V),$   
 $\quad \quad \quad (\text{all\_to\_right}_{k+1} \ V \ N \ (\text{pred}_k \ X)).$

Similarly we define the predicate  $\text{non\_zero}_{k+1}$  that succeeds if its argument is not equal to  $\text{zero}_{k+1}$ :

$\text{non\_zero}_{k+1} \ N \quad \leftarrow \quad (\text{exists\_to\_right}_{k+1} \ \text{high} \ N \ \text{last}_k).$   
 $\text{exists\_to\_right}_{k+1} \ V \ N \ X \quad \leftarrow \quad (N \ X \ V).$   
 $\text{exists\_to\_right}_{k+1} \ V \ N \ X \quad \leftarrow \quad (\text{non\_zero}_k \ X),$   
 $\quad \quad \quad (\text{exists\_to\_right}_{k+1} \ V \ N \ (\text{pred}_k \ X)).$

Symmetrically, we define the predicates  $\text{is\_last}_{k+1}$  and  $\text{non\_last}_{k+1}$ , as follows:

$\text{is\_last}_{k+1} \ N \quad \leftarrow \quad (\text{all\_to\_right}_{k+1} \ \text{high} \ N \ \text{last}_k).$   
 $\text{non\_last}_{k+1} \ N \quad \leftarrow \quad (\text{exists\_to\_right}_{k+1} \ \text{low} \ N \ \text{last}_k).$

Using the above predicates we can now define  $\text{pred}_{k+1}$  as follows:

$\text{pred}_{k+1} \ N \ X \ V \quad \leftarrow \quad (\text{is\_zero}_k \ X), (\text{non\_zero}_{k+1} \ N),$   
 $\quad \quad \quad (N \ X \ V1), (\text{invert} \ V1 \ V).$   
 $\text{pred}_{k+1} \ N \ X \ V \quad \leftarrow \quad (\text{non\_zero}_k \ X),$   
 $\quad \quad \quad (\text{exists\_to\_right}_{k+1} \ \text{high} \ N \ (\text{pred}_k \ X)), (N \ X \ V).$   
 $\text{pred}_{k+1} \ N \ X \ V \quad \leftarrow \quad (\text{non\_zero}_k \ X), (\text{non\_zero}_{k+1} \ N),$   
 $\quad \quad \quad (\text{all\_to\_right}_{k+1} \ \text{low} \ N \ (\text{pred}_k \ X)),$   
 $\quad \quad \quad (N \ X \ V1), (\text{invert} \ V1 \ V).$

In a symmetric way we define the predicate  $\text{succ}_{k+1}$  as follows:

$\text{succ}_{k+1} \ N \ X \ V \quad \leftarrow \quad (\text{is\_zero}_k \ X), (\text{non\_last}_{k+1} \ N),$   
 $\quad \quad \quad (N \ X \ V1), (\text{invert} \ V1 \ V).$   
 $\text{succ}_{k+1} \ N \ X \ V \quad \leftarrow \quad (\text{non\_zero}_k \ X),$   
 $\quad \quad \quad (\text{exists\_to\_right}_{k+1} \ \text{low} \ N \ (\text{pred}_k \ X)), (N \ X \ V).$   
 $\text{succ}_{k+1} \ N \ X \ V \quad \leftarrow \quad (\text{non\_zero}_k \ X), (\text{non\_zero}_{k+1} \ N),$   
 $\quad \quad \quad (\text{all\_to\_right}_{k+1} \ \text{high} \ N \ (\text{pred}_k \ X)),$   
 $\quad \quad \quad (N \ X \ V1), (\text{invert} \ V1 \ V).$

Equality of two numbers is defined as follows:

$\text{equal}_{k+1} \ N \ M \quad \leftarrow \quad (\text{equal\_test}_{k+1} \ N \ M \ \text{last}_k).$   
 $\text{equal\_test}_{k+1} \ N \ M \ X \quad \leftarrow \quad (\text{is\_zero}_k \ X), (N \ X \ V), (M \ X \ V).$   
 $\text{equal\_test}_{k+1} \ N \ M \ X \quad \leftarrow \quad (\text{non\_zero}_k \ X), (N \ X \ V), (M \ X \ V),$   
 $\quad \quad \quad (\text{equal\_test}_{k+1} \ N \ M \ (\text{pred}_k \ X)).$

Finally,  $\text{less\_than}_{k+1}$  can be defined in an identical way as in the previous subsection.

### 4.3 Simulating Turing Machines with Higher-Order Datalog

In this section we demonstrate how we can use  $k$ -order Datalog to simulate  $(k - 1)$ -exponential-time-bounded Turing machines. The simulation using second-order programs does not differ from the one that uses programs of order greater than or equal to three, except for a minor difference described below.

In order to define the initialization rules for the Turing machine, we will need a predicate that transforms the numbers  $0, \dots, n-1$  that appear in the **input** relation, to order- $k$  representation of numbers. Recall that such a number is a function from order- $(k - 1)$  numbers to the values **low** and **high**:

```
base_to_higher_k 0 X low.
base_to_higher_k M X V ← (input J σ M), (succ_k (base_to_higher_k J) X V).
```

When  $k = 2$  the only required change in the above predicate is to replace all the occurrences of  $X$  by  $\bar{X}$ .

The simulation of the exponential-time bounded Turing machine is presented below. Recall that we assume that in the beginning of its operation, the first  $n$  squares of the tape hold the input, the rest of the squares hold the empty character “ $\_$ ” and the machine starts operating from its initial state denoted by  $s_0$ . If the Turing machine accepts the input then it goes into the special state called **yes** and remains in this state forever.

One new feature (with respect to the first-order case), is the representation of the **cursor** predicate. In the first-order case we used **cursor**  $\bar{T}$   $\bar{X}$  to mean that at time-point  $\bar{T}$  the cursor is located at position  $\bar{X}$ . In the higher-order case the expression (**cursor**  $T$ ) is a number that denotes the position of the cursor, ie., it is a function from positions to the values **low** and **high**. We start with the initialization rules:

```
symbol_σ T X ← (is_zero_k T), (input Y σ W),
               equal_k (base_to_higher_k Y) X.
symbol_⊥ T X ← (is_zero_k T), (base_last Y),
               (less_than_k (base_to_higher_k Y) X).
state_{s_0} T ← (is_zero_k T).
cursor T I low ← (is_zero_k T).
```

We now define the transition rules of the Turing machine. For each transition rule we generate rules for **state** <sub>$s$</sub> , **symbol** <sub>$\sigma$</sub>  and **cursor**. The transition: “if the head is in symbol  $\sigma$  and in state  $s$  then write symbol  $\sigma'$  and go to state  $s'$ ”, generates the following:

```
symbol_{σ'} T X ← (non_zero_k T), (equal_k X (cursor (pred_k T))),
                  (state_s (pred_k T)),
                  (symbol_σ (pred_k T) (cursor (pred_k T))).
state_{s'} T ← (non_zero_k T), (state_s (pred_k T)),
               (symbol_σ (pred_k T) (cursor (pred_k T))).
cursor T I V ← (non_zero_k T), (state_s (pred_k T)),
               (symbol_σ (pred_k T) (cursor (pred_k T))),
               (cursor (pred_k T) I V).
```

We continue with the transition: “if the head is in symbol  $\sigma$  and in state  $s$  then go to state  $s'$  and move the head right”, which generates the following:

$$\begin{aligned}
\text{symbol}_\sigma \ T \ X &\leftarrow (\text{non\_zero}_k \ T), (\text{equal}_k \ X \ (\text{cursor} \ (\text{pred}_k \ T))), \\
&\quad (\text{state}_s \ (\text{pred}_k \ T)), \\
&\quad (\text{symbol}_\sigma \ (\text{pred}_k \ T) \ (\text{cursor} \ (\text{pred}_k \ T))). \\
\text{state}_{s'} \ T &\leftarrow (\text{non\_zero}_k \ T), (\text{state}_s \ (\text{pred}_k \ T)), \\
&\quad (\text{symbol}_\sigma \ (\text{pred}_k \ T) \ (\text{cursor} \ (\text{pred}_k \ T))). \\
\text{cursor} \ T \ I \ V &\leftarrow (\text{non\_zero}_k \ T), (\text{state}_s \ (\text{pred}_k \ T)), \\
&\quad (\text{symbol}_\sigma \ (\text{pred}_k \ T) \ (\text{cursor} \ (\text{pred}_k \ T))), \\
&\quad ((\text{succ}_k \ (\text{cursor} \ (\text{pred}_k \ T))) \ I \ V).
\end{aligned}$$

We also have the transition: “if the head is in symbol  $\sigma$  and in state  $s$  then go to state  $s'$  and move the head left”, which generates the following:

$$\begin{aligned}
\text{symbol}_\sigma \ T \ X &\leftarrow (\text{non\_zero}_k \ T), (\text{equal}_k \ X \ (\text{cursor} \ (\text{pred}_k \ T))), \\
&\quad (\text{state}_s \ (\text{pred}_k \ T)), \\
&\quad (\text{symbol}_\sigma \ (\text{pred}_k \ T) \ (\text{cursor} \ (\text{pred}_k \ T))). \\
\text{state}_{s'} \ T &\leftarrow (\text{non\_zero}_k \ T), (\text{state}_s \ (\text{pred}_k \ T)), \\
&\quad (\text{symbol}_\sigma \ (\text{pred}_k \ T) \ (\text{cursor} \ (\text{pred}_k \ T))). \\
\text{cursor} \ T \ I \ V &\leftarrow (\text{non\_zero}_k \ T), (\text{state}_s \ (\text{pred}_k \ T)), \\
&\quad (\text{symbol}_\sigma \ (\text{pred}_k \ T) \ (\text{cursor} \ (\text{pred}_k \ T))), \\
&\quad ((\text{pred}_k \ (\text{cursor} \ (\text{pred}_k \ T))) \ I \ V).
\end{aligned}$$

The inertia rules are the following:

$$\begin{aligned}
\text{symbol}_\sigma \ T \ X &\leftarrow (\text{less\_than}_k \ X \ (\text{cursor} \ (\text{pred}_k \ T))), (\text{symbol}_\sigma \ (\text{pred}_k \ T) \ X). \\
\text{symbol}_\sigma \ T \ X &\leftarrow (\text{less\_than}_k \ (\text{cursor} \ (\text{pred}_k \ T)) \ X), (\text{symbol}_\sigma \ (\text{pred}_k \ T) \ X).
\end{aligned}$$

Finally, we have the following rule that concerns acceptance:

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

When  $k = 2$  the only required change in the clauses given above, is to replace all the occurrences of  $I$  in the definition of `cursor` by  $\bar{I}$ .

## 5 Future Work

Theorem 3 presents a striking analogy with Theorem 7.17 of (Jones 2001) where it is shown that read-only functional programs of order  $k \geq 2$  capture  $(k - 1)$ -EXPTIME, and first-order such programs capture PTIME. The functional language that Jones uses has no direct relationship with Datalog, and this makes the analogy even more interesting.

As a possible direction for future research, we would like to investigate whether the complexity results obtained in (Jones 2001) regarding *tail-recursive* read-only functional programs, can extend to the case of Higher-Order Datalog. A starting point for this would be to first characterize what tail-recursion means in the context of Higher-Order Datalog. An additional topic for future research would be to investigate the complexity-theoretic benefits of adding negation to Higher-Order Datalog. Such an investigation can be based on the recent proposal for the well-founded semantics of higher-order logic programs (Charalambidis et al. 2018).

## Acknowledgements

The research of the first author was supported by the Software and Knowledge Engineering Laboratory (SKEL) of the Institute of Informatics and Telecommunications of NCSR “Demokritos”.



## Supplementary material

To view supplementary material for this article, please visit <https://doi.org/10.1017/S1471068419000279>.

## References

- BEZEM, M. 1999. Extensionality of simply typed logic programs. In *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, D. D. Schreye, Ed. MIT Press, 395–410.
- CHARALAMBIDIS, A., HANDJOPOULOS, K., RONDOGIANNIS, P., AND WADGE, W. W. 2013. Extensional higher-order logic programming. *ACM Trans. on Computational Logic* 14, 3, 21.
- CHARALAMBIDIS, A., RONDOGIANNIS, P., AND SYMEONIDOU, I. 2018. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs. *TPLP* 18, 3-4, 421–437.
- CHARALAMBIDIS, A., RONDOGIANNIS, P., AND TROUMPOUKIS, A. 2018. Higher-order logic programming: An expressive language for representing qualitative preferences. *Science of Computer Programming* 155, 173–197.
- CHEN, W., KIFER, M., AND WARREN, D. S. 1993. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming* 15, 3, 187–230.
- DANTSIN, E., EITER, T., GOTTLÖB, G., AND VORONKOV, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys* 33, 3, 374–425.
- 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.
- KOUNTOURIOTIS, V., RONDOGIANNIS, P., AND WADGE, W. W. 2005. Extensional higher-order datalog. In *Short Paper Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. 1–5.
- LEIVANT, D. 1989. Descriptive characterizations of computational complexity. *Journal of Computer and System Science* 39, 1, 51–83.
- LOYD, J. W. 1987. *Foundations of Logic Programming*. Springer Verlag.
- LOVRENČIĆ, A. AND ČUBRILO, M. 1999. Amalgamation of heterogeneous data sources using amalgamated annotated hilog. In *3rd international IEEE Conference on Intelligent Engineering Systems (INES'99)*.
- MILLER, D. AND NADATHUR, G. 1986. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming (ICLP)*. 448–462.
- 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, San Francisco, California, USA*. ACM, 137–146.
- WADGE, W. W. 1991. Higher-order horn logic programming. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*. MIT Press, 289–303.
- YANG, G., KIFER, M., AND ZHAO, C. 2003. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*. Vol. 2888. Springer, 671–688.