Online appendix for the paper

The Expressive Power of Higher-Order Datalog

published in Theory and Practice of Logic Programming

ANGELOS CHARALAMBIDIS
Institute of Informatics and Telecommunications, NCSR "Demokritos", Greece
(*e-mail:* acharal@iit.demokritos.gr)

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

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

Appendix A The Expressive Power of First-Order Datalog

In this appendix we present a proof of the well-known theorem (Papadimitriou 1985; Grädel 1992; Vardi 1982; Immerman 1986; Leivant 1989) that Datalog captures PTIME (under the assumption that the input strings are encoded, as already discussed, through the `input` relation). Our proof builds on that of (Papadimitriou 1985) and (Dantsin et al. 2001), but gives more technical details.

Theorem 2
The set of first-order Datalog programs captures PTIME.

Proof
The proof of the above theorem consists of the proofs of the following two statements:

Statement 1
Every language L decided by a Datalog program P, can also be decided by a Turing machine in time $O(n^q)$, where n is the length of the input string and q is a constant that depends only on P.

Statement 2
Every language L decided by a Turing machine in time $O(n^q)$, where n is the length of its input, can be decided by a Datalog program P.

Proof of Statement 1
Assume that the maximum number of atoms that appear in any rule in P is equal to l, the total number of constants that appear in P is equal to c (including `a`, `b`, `empty`, `end`, but excluding the n natural numbers that appear in the relation `input`), the total number of rules in P is equal to r, the total number of predicates is equal to p, and the maximum arity of a predicate that appears in P is t. We present a multi-tape Turing machine which decides the language L in time $O(n^q)$, where n is the length of the input string, for some q that depends only on the above characteristics of P.

The Turing machine, with input w, starts by constructing the set of facts $\mathcal{D}_w$ that represent w in the Datalog program (i.e., the relation `input`), which are stored on a separate tape. Each number that appears as an argument in the relation `input` is written in binary using $O(\log n)$ bits. The construction of $\mathcal{D}_w$ requires $O(n \cdot \log n)$ time.

Next, the Turing machine executes the usual bottom-up procedure for computing the least fixed-point of a Datalog program through the iterations of the T_{P} operator. Intuitively, it starts by assigning the empty relation to all predicates in P and at each iteration it examines each clause of P and determines if it can generate any new tuples. The relations assigned to predicates in P are stored each on a separate tape. Observe that there are at most $p \cdot (n+c)^t$ tuples in the minimum Herbrand model M_{P} of P (in the extreme case where all the predicates have the same maximum arity t and all possible tuples for all possible predicates belong to the minimum model). Therefore, the bottom-up procedure will terminate after at most $p \cdot (n+c)^t$ iterations, since at each iteration at least one tuple must be produced. Each such iteration of the bottom-up computation takes polynomial time with respect to n:

- For every rule, the machine instantiates all the variables using the $(n+c)$ available constants. The number of different such instantiations of a rule is bounded by $(n+c)^{l \cdot t}$.
- For each such instantiation it examines if the atoms in the body of the rule have already been produced in a previous step of the computation. Searching through the list of the already produced atoms for a specific predicate takes time $O(t \cdot \log n \cdot (n+c)^t)$ in the worst case (since the maximum number of atoms that such a list may contain is $(n+c)^t$ and the length of each atom is $O(t \cdot \log n)$). Doing this for all atoms in the rule body, requires time $O(l \cdot t \cdot \log n \cdot (n+c)^t)$. If all atoms in the (instantiated) body of the rule are found in the corresponding lists, then we search the head of the rule in the list that corresponds to its predicate; if it is not found, then it is appended at the end of the list. This search and update requires time $O(t \cdot \log n \cdot (n+c)^t)$.
- Doing the above operation for all the rules of the program requires time $O(r \cdot l \cdot t \cdot \log n \cdot (n+c)^{(l+1) \cdot t})$.

From the above we get that in order to produce the minimum Herbrand model M_{P} of P, we need time $O(n \cdot \log n + p \cdot r \cdot l \cdot t \cdot \log n \cdot (n+c)^{(l+2) \cdot t})$. Since p, r, l, t and c are constants that depend only on P and do not depend on n, the running time of the Turing machine is $O(n^q)$ for $q = (l+2) \cdot t$.

The Turing machine returns *yes* if and only if `accept` is true in the minimum Herbrand model M_{P}.

Proof of Statement 2

In order to establish the second statement, we need to define a simulator of the Turing machine in Datalog. Assume that M decides L in time $O(n^q)$. Then there exists an integer constant d, such that for every input w of length $n \geq 2$, M terminates after at most $n^d - 1$ steps. The Datalog program that is presented below produces the correct answer for all inputs of length at least 2 by simulating $n^d - 1$ steps of the Turing machine M. For the special cases of strings of length 0 or 1 that belong to L, the correct answer

is produced directly by appropriate rules (notice that for $n = 1$, the value of $n^d - 1$ is 0, regardless of the choice of d).

We start by defining predicates `base_zero` (which is true of 0, ie. of the first argument of the first tuple in the `input` relation), `base_last` (which is true of $n-1$, ie., of the first argument of the *last* tuple in the `input` relation), `base_succ` (which, given a number k, $0 \leq k < n-1$, returns $k+1$), and `base_pred` (which given $k+1$ returns k):

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

Given the above predicates, we can simulate counting from 0 up to $n-1$. We extend the range of the numbers we can support up to $n^d - 1$ for any fixed d by using d distinct arguments in predicates; we view these d arguments more conveniently as d-tuples, ie., we use the notation $\overline{\mathtt{X}}$ to represent the sequence of d arguments $\mathtt{X}_1, \ldots, \mathtt{X}_d$. We define the predicates `tuple_zero`, `tuple_last` and `tuple_base_last` that act on such d-tuples and represent the numbers 0, $n^d - 1$ and $n - 1$ respectively.

```
tuple_zero X̄       ←  (base_zero X_1),...,(base_zero X_d).
tuple_last X̄       ←  (base_last X_1),...,(base_last X_d).
tuple_base_last X̄  ←  (base_zero X_1),...,(base_zero X_{d-1}),
                      (base_last X_d).
```

To define `tuple_succ` we need d clauses that have as arguments two tuples having d elements each:

```
tuple_succ X̄ Ȳ  ←  (X_1 ≈ Y_1),...,(X_{d-1} ≈ Y_{d-1}),
                   (base_succ X_d Y_d).
tuple_succ X̄ Ȳ  ←  (X_1 ≈ Y_1),...,(X_{d-2} ≈ Y_{d-2}),
                   (base_succ X_{d-1} Y_{d-1}),
                   (base_last X_d),
                   (base_zero Y_d).
                ...
tuple_succ X̄ Ȳ  ←  (base_succ X_1 Y_1),
                   (base_last X_2),...,(base_last X_d),
                   (base_zero Y_2),...,(base_zero Y_d).
```

Now we can easily define `tuple_pred` as follows:

```
tuple_pred X̄ Ȳ  ←  tuple_succ Ȳ X̄.
```

The `less_than` relation over the numbers we consider, is defined as follows:

```
less_than X̄ Ȳ  ←  tuple_succ X̄ Ȳ.
less_than X̄ Ȳ  ←  (tuple_succ X̄ Z̄),(less_than Z̄ Ȳ).
```

We can also define `tuple_non_zero`, namely the predicate that succeeds if its argument is not equal to zero:

```
tuple_non_zero X̄  ←  (tuple_zero Z̄),(less_than Z̄ X̄).
```

We now define predicates $\texttt{symbol}_\sigma$, $\texttt{state}_s$ and $\texttt{cursor}$, for every $\sigma \in \Sigma$ and for every state s of the Turing machine we are simulating. Intuitively, $\texttt{symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}$ succeeds if the tape has symbol σ in position $\overline{\texttt{X}}$ of the tape at time-step $\overline{\texttt{T}}$, $\texttt{state}_s\ \overline{\texttt{T}}$ succeeds if the machine is in state s at step $\overline{\texttt{T}}$ and $\texttt{cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}$ succeeds if the head of the machine points at position $\overline{\texttt{X}}$ at step $\overline{\texttt{T}}$. Since symbols and states are finite there will be a finite number of clauses defining the above predicates. We assume that the Turing machine never attempts to go to the left of its leftmost symbol. Moreover, 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 stays there forever.

The initialization of the Turing machine is performed by the following clauses:

$$
\begin{array}{lcl}
\texttt{symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_zero}\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(base\_zero}\ \texttt{X}_1\texttt{)},\ldots,\texttt{(base\_zero}\ \texttt{X}_{d-1}\texttt{)}, \\
 & & \texttt{(input}\ \texttt{X}_d\ \sigma\ \texttt{W)}. \\
\texttt{symbol}_\sqcup\ \overline{\texttt{T}}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_zero}\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(tuple\_base\_last}\ \overline{\texttt{Y}}\texttt{),(less\_than}\ \overline{\texttt{Y}}\ \overline{\texttt{X}}\texttt{)}. \\
\texttt{state}_{s_0}\ \overline{\texttt{T}} & \leftarrow & \texttt{(tuple\_zero}\ \overline{\texttt{T}}\texttt{)}. \\
\texttt{cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_zero}\ \overline{\texttt{T}}\texttt{),(tuple\_zero}\ \overline{\texttt{X}}\texttt{)}.
\end{array}
$$

For each transition rule we generate a set of clauses. We start with the rule "if the head is in symbol σ and in state s then write symbol σ' and go to state s'", which is translated as follows:

$$
\begin{array}{lcl}
\texttt{symbol}_{\sigma'}\ \overline{\texttt{T}'}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{)}. \\
\texttt{state}_{s'}\ \overline{\texttt{T}'} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{)}. \\
\texttt{cursor}\ \overline{\texttt{T}'}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{)}.
\end{array}
$$

We continue with the transition: "if the head is in symbol σ and in state s, then go to state s' and move the head right", which generates the following:

$$
\begin{array}{lcl}
\texttt{symbol}_\sigma\ \overline{\texttt{T}'}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{)}. \\
\texttt{state}_{s'}\ \overline{\texttt{T}'} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{)}. \\
\texttt{cursor}\ \overline{\texttt{T}'}\ \overline{\texttt{X}'} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(tuple\_succ}\ \overline{\texttt{X}}\ \overline{\texttt{X}'}\texttt{)}.
\end{array}
$$

We also have the transition: "if the head is in symbol σ and in state s then go to state s' and move the head left", which generates the following:

$$
\begin{array}{lcl}
\texttt{symbol}_\sigma\ \overline{\texttt{T}'}\ \overline{\texttt{X}} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{)}. \\
\texttt{state}_{s'}\ \overline{\texttt{T}'} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{)}. \\
\texttt{cursor}\ \overline{\texttt{T}'}\ \overline{\texttt{X}'} & \leftarrow & \texttt{(tuple\_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}\texttt{),(state}_s\ \overline{\texttt{T}}\texttt{)}, \\
 & & \texttt{(cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}}\texttt{),(tuple\_pred}\ \overline{\texttt{X}}\ \overline{\texttt{X}'}\texttt{)}.
\end{array}
$$

We also need to provide "inertia" rules for the tape squares that are not affected by the above rules. These squares are exactly those that have a different position from the one pointed to by the cursor. Therefore, the following clauses will suffice:

$$\begin{aligned}
\texttt{symbol}_\sigma\ \overline{\texttt{T}'}\ \overline{\texttt{X}'} \quad &\leftarrow \quad (\texttt{tuple_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}),(\texttt{cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}),\\
&\qquad\quad (\texttt{less_than}\ \overline{\texttt{X}}\ \overline{\texttt{X}'}),(\texttt{symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}'}).\\
\texttt{symbol}_\sigma\ \overline{\texttt{T}'}\ \overline{\texttt{X}'} \quad &\leftarrow \quad (\texttt{tuple_succ}\ \overline{\texttt{T}}\ \overline{\texttt{T}'}),(\texttt{cursor}\ \overline{\texttt{T}}\ \overline{\texttt{X}}),\\
&\qquad\quad (\texttt{less_than}\ \overline{\texttt{X}'}\ \overline{\texttt{X}}),(\texttt{symbol}_\sigma\ \overline{\texttt{T}}\ \overline{\texttt{X}'}).
\end{aligned}$$

Lastly, the following rule succeeds iff the Turing machine succeeds after $n^d - 1$ steps.

$$\texttt{accept} \quad \leftarrow \quad (\texttt{tuple\_last}\ \overline{\texttt{T}}),(\texttt{state}_{\texttt{yes}}\ \overline{\texttt{T}}).$$

In the case of strings of length $n \leq 1$ that belong to L, we add appropriate rules to the Datalog program. For example, if $a \in L$, then the following rule is included in the Datalog program:

$$\texttt{accept} \quad \leftarrow \quad (\texttt{input 0 a end}).$$

This completes the proof of the theorem. □

Appendix B Proof of Lemma 1

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 the same language in time $O(\exp_{k-1}(n^q))$, where n is the length of the input string and q is a constant that depends only on P.

Proof
Let P be a k-order Datalog program that decides a language L. Assume that the maximum length of a rule in P is equal to l, the total number of constants that appear in P is equal to c (including `a`, `b`, `empty`, `end`, but excluding the n natural numbers that appear in the relation `input`), the total number of rules in P is equal to r, the total number of predicates is equal to p, the total number of predicates types involved in P is equal to s, and the maximum arity of a predicate type that is involved in P is t. A summary of all these parameters is given in Table B1. We present a multi-tape Turing machine which

Symbol	*Characteristic of* P
l	maximum length of a rule
c	number of constants
r	number of rules
p	number of predicates
s	number of predicate types
t	maximum arity of a predicate type

Table B 1. *Characteristics of* P *used in our analysis.*

decides the language L in time $O(\exp_{k-1}(n^q))$, where n is the length of the input string, for some q that depends only on the above characteristics of P.

The Turing machine, with input w, starts by constructing the set of facts $\mathcal{D}_w$ that represent w in the Datalog program (i.e., the relation `input`), which are stored on a separate tape. Each number that appears as an argument in the relation `input` is written in binary using $O(\log n)$ bits. It also writes on a separate tape all the elements of the set $[\![\iota]\!]$ (that is, the c constants that occur in P and the n numbers that appear in the input relation). This requires $O(n \cdot \log n)$ time.

Subsequently, the Turing machine performs two major phases: (i) it produces all the monotonic relations that are needed for the bottom-up execution of the program, and (ii) it performs instantiations of the rules using these monotonic relations as-well-as individual constants, computing in this way, in a bottom-up manner, the minimum Herbrand model of P. The complexity of these two major phases is analyzed in detail below.

Complexity of producing the monotonic relations. The Turing machine constructs the set $[\![\rho]\!]$, for every predicate type $[\![\rho]\!]$ of order at most $k-1$, which is involved in P. The elements of $[\![\rho]\!]$ are monotonic functions, which are represented by their corresponding relations. Predicate types are considered in increasing order, and for each type $[\![\rho]\!]$ the set $[\![\rho]\!]$ is stored on a separate tape. These sets will be used later by the Turing machine, each time that it needs to instantiate predicate variables that occur in the rules of P.

Before we present in more details the above construction and analyze its time complexity, we need to calculate upper bounds for the number of elements in $[\![\rho]\!]$ and for the length of their representation. We prove that, for every j-order predicate ρ, the number of elements in $[\![\rho]\!]$ is at most $\exp_j(t^{j-1} \cdot (n+c)^t)$ and each of them can be represented using $O(\log n \cdot \exp_{j-1}(j \cdot t^j \cdot (n+c)^t))$ symbols. These two statements can be proved simultaneously by induction on j, as shown below.

For the basis of the induction, consider a first order predicate type ρ of arity $m \leq t$. Each element in $[\![\rho]\!]$ corresponds to a set of tuples, where each tuple consists of m constants. Since there are $(n+c)$ different constants in $\mathsf{P} \cup \mathcal{D}_w$, there are $2^{(n+c)^m} \leq 2^{(n+c)^t} = \exp_1(t^0 \cdot (n+c)^t)$ elements in $[\![\rho]\!]$. Moreover, every element in $[\![\rho]\!]$ contains at most $(n+c)^t$ tuples, each tuple consists of at most t constants and each constant can be represented using $O(\log n)$ symbols. Thus the length of the representation of every element in $[\![\rho]\!]$ is $O(\log n \cdot t \cdot (n+c)^t) = O(\log n \cdot \exp_0(1 \cdot t^1 \cdot (n+c)^t))$. Thus, the statement holds for $j = 1$.

For the induction step, assume that $j > 1$ and that our statement holds for all $i < j$. Consider a j-order predicate type $\rho = \rho_1 \to \cdots \to \rho_m \to o$ of arity $m \leq t$. Each element in $[\![\rho]\!]$ corresponds to a subset of $[\![\rho_1]\!] \times \cdots \times [\![\rho_m]\!]$. For every ν, $1 \leq \nu \leq m$, ρ_ν is either equal to ι, or is an i-order predicate type, with $i < j$. In the former case, $[\![\rho_\nu]\!]$ contains $(n+c)$ elements; in the latter case $[\![\rho_\nu]\!]$ contains at most $\exp_i(t^{i-1} \cdot (n+c)^t)$ elements, by the induction hypothesis. In both cases $[\![\rho_\nu]\!]$ contains at most $\exp_{j-1}(t^{j-2} \cdot (n+c)^t)$ elements. Using some properties of the function exp, we get that the number of elements in $[\![\rho]\!]$ is bounded by $2^{(\exp_{j-1}(t^{j-2} \cdot (n+c)^t))^m} \leq 2^{\exp_{j-1}(m \cdot t^{j-2} \cdot (n+c)^t)} \leq 2^{\exp_{j-1}(t^{j-1} \cdot (n+c)^t)} = \exp_j(t^{j-1} \cdot (n+c)^t)$. Moreover, every element in $[\![\rho]\!]$ corresponds to a relation that contains at most $(\exp_{j-1}(t^{j-2} \cdot (n+c)^t))^t \leq \exp_{j-1}(t^{j-1} \cdot (n+c)^t)$ tuples; each one of these tuples consists of at most t elements and, by the induction hypothesis, each element can be represented using $O(\log n \cdot \exp_{j-2}((j-1) \cdot t^{j-1} \cdot (n+c)^t))$ symbols. By the properties of the

function exp it follows that $t \cdot \exp_{j-2}((j-1) \cdot t^{j-1} \cdot (n+c)^t) \leq \exp_{j-2}((j-1) \cdot t^j \cdot (n+c)^t) \leq \exp_{j-1}((j-1) \cdot t^j \cdot (n+c)^t)$ and $\exp_{j-1}(t^{j-1} \cdot (n+c)^t) \cdot \exp_{j-1}((j-1) \cdot t^j \cdot (n+c)^t) \leq \exp_{j-1}(j \cdot t^j \cdot (n+c)^t)$. Thus, the length of the representation of a j-order relation is $O(\log n \cdot \exp_{j-1}(j \cdot t^j \cdot (n+c)^t))$.

Since c, t and j are constants that do not depend on n, it follows that for every j-order predicate ρ, the number of elements in $[\![\rho]\!]$ is $O(\exp_j(n^{t+1}))$ and each of these elements can be represented using $O(\exp_{j-1}(n^{t+1}))$ symbols.

In order to create a list with all the elements of type $[\![\rho]\!]$ for a j-order type $\rho = \rho_1 \to \cdots \to \rho_m \to o$ of arity $m \leq t$, the Turing machine first constructs the cartesian product $S = [\![\rho_1]\!] \times \cdots \times [\![\rho_m]\!]$. Since $\rho_1, \cdots, \rho_m$ have order at most $j-1$, the sets $[\![\rho_1]\!], \cdots, [\![\rho_m]\!]$ have already been constructed in previous steps of the Turing machine. The construction of S requires time linear to the length of its representation, provided that the sets $[\![\rho_1]\!], \cdots, [\![\rho_m]\!]$ are stored on separate tapes. Since ρ may have arguments of the same type, this may require to create at most $m-1$ copies of such sets. Notice that the sets $S, [\![\rho_1]\!], \cdots, [\![\rho_m]\!]$ are actually j-order relations, and therefore their representations have length $O(\exp_{j-1}(n^{t+1}))$. We conclude that the time required to create S is $O(\exp_{j-1}(n^{t+1}))$ (since m is a constant that does not depend on n).

The set $[\![\rho]\!]$ contains the elements in the powerset 2^S of S which represent monotonic functions. The set 2^S can be constructed in time linear to the length of its representation. Since 2^S is a $(j+1)$-order relation, this length is $O(\exp_j(n^{t+1}))$. Thus, 2^S can be constructed in time $O(\exp_j(n^{t+1}))$. Now, $[\![\rho]\!]$ can be obtained from 2^S, by removing elements that represent non-monotonic functions. In order to decide whether a relation in 2^S belongs to $[\![\rho]\!]$, it suffices to consider each pair of elements in S and verify that, for this pair, the monotonicity property is not violated. This verification requires time linear to the length of the relation, for each pair of elements in S. Thus, the time required to check whether a relation represents a monotonic function is $O((\exp_{j-1}(n^{t+1}))^{2t} \cdot \exp_{j-1}(n^{t+1}))$. The number of relations in 2^S is $O(\exp_j(n^{t+1}))$. Using the properties of the function exp, we get that $(\exp_{j-1}(n^{t+1}))^{2t} \cdot \exp_{j-1}(n^{t+1}) \cdot \exp_j(n^{t+1}) \leq \exp_j(2t \cdot n^{t+1}) \cdot \exp_j(n^{t+1}) \cdot \exp_j(n^{t+1}) \leq \exp_j((2t+2) \cdot n^{t+1}) \leq \exp_j(n^{t+2})$. Thus, the removal of relations that do not correspond to monotonic functions requires time $O(\exp_j(n^{t+2}))$.

By adding the times required to construct S and 2^S, and remove non-relevant relations, we conclude that the time to create a list with all the elements in $[\![\rho]\!]$ is $O(\exp_{j-1}(n^{t+1})) + O(\exp_j(n^{t+1})) + O(\exp_j(n^{t+2})) = O(\exp_j(n^{t+2}))$.

The above process is executed for each of the s predicate types of order at most $k-1$ involved in P. Since s is a constant that does not depend on n, the time required for the construction of all the relations of each predicate type is $O(\exp_{k-1}(n^{t+2}))$.

Complexity of performing the bottom-up computation. Next, the Turing machine essentially computes the successive approximations to the minimum Herbrand model M_P of P, by iterative application of the T_P operator (as described at the end of Section 2). For each predicate constant defined in P, the relation that represents its meaning is written on a separate tape; initially all these relations are empty. At each iteration of the T_P operator, new tuples may be added to the meaning of predicate constants. In order to calculate one iteration of T_P, the Turing machine considers each clause in P and examines what new tuples it can produce. More specifically, given a rule, it replaces every individual variable that appears in the rule by a constant symbol and every predicate

variable with a monotonic relation of the same type as the variable; moreover, it replaces every predicate constant, say q, that appears in the body of the clause with the relation that has been computed for q during the previous iterations of T_{P}. It then checks if the body of the instantiated clause evaluates to true: this is performed by essentially checking if elements belong to sets. If an instantiation of a clause body evaluates to true, the instantiated head is added to the meaning of the head predicate.

Observe that there are at most $p \cdot (\exp_{k-1}(t^{k-2} \cdot (n+c)^t))^t$ tuples in the minimum Herbrand model M_{P} of P (in the extreme case, all the predicates have order k, the same maximum arity t and all possible tuples for all possible predicates belong to the minimum model). Therefore, the bottom-up procedure will terminate after at most $p \cdot (\exp_{k-1}(t^{k-2} \cdot (n+c)^t))^t$ iterations, since at each iteration at least one tuple must be produced. Since $(\exp_{k-1}(t^{k-2} \cdot (n+c)^t))^t \leq \exp_{k-1}(t^{k-1} \cdot (n+c)^t)$ and k, p, t are constants that do not depend on n, the number of iterations is $O(\exp_{k-1}(n^{t+1}))$.

We calculate a bound of the time that is required for each one of the above iterations:

- For every rule in the program, the Turing machine instantiates each individual variable in the rule using elements in $[\![\iota]\!]$. Moreover, it instantiates each predicate variables of type ρ with relations representing monotonic functions in $[\![\rho]\!]$ (recall that these sets have been constructed in the first phase of the execution of the Turing machine). Finally, it replaces every predicate constant in the body of the rule with the relation that has already been computed for it during the previous iterations of T_{P}. By the syntactic rules of Higher-Order Datalog programs, predicate variables may have order at most $k-1$. Thus, the number of different such instantiations of a rule is bounded by $(\exp_{k-1}(t^{k-2} \cdot (n+c)^t))^l$. Since $(\exp_{k-1}(t^{k-2} \cdot (n+c)^t))^l \leq \exp_{k-1}(l \cdot t^{k-2} \cdot (n+c)^t)$ and k, l, t are constants that do not depend on n, the number of different instantiations for each rule is $O(\exp_{k-1}(n^{t+1}))$.
- Each rule contains a constant number of (individual or predicate) variables. Since all predicate variables have order at most $k-1$, each variable is replaced by at most $O(\exp_{k-2}(n^{t+1}))$ symbols. Thus, the length of the instantiated rule is $O(\exp_{k-2}(n^{t+1}))$. Moreover, the instantiation can be computed in time linear to its length.
- For each such instantiation the Turing machine examines if the body of the rule evaluates to *true*. This may require at most l rewritings of the instantiated body of the rule, each resulting after partially applying a predicate to its first argument. Each rewriting requires time linear to the length of the instantiated rule, that is, $O(\exp_{k-2}(n^{t+1}))$. Since l does not depend on n, the total time that is needed to examine if the body of the rule evaluates to *true* is $O(\exp_{k-2}(n^{t+1}))$.
- If the body of some instantiated rule evaluates to *true*, then we search the head of the rule in the list that corresponds to its predicate; if it is not found, then it is inserted in the list. This search and insertion requires time linear to the length of this list, which is $O(\exp_{k-1}(n^{t+1}))$.
- The total time needed to repeat the above process for every instantiation of a specific rule is $O(\exp_{k-1}(n^{t+1})) \cdot O(\exp_{k-1}(n^{t+1})) = O((\exp_{k-1}(n^{t+1}))^2)$.
- Since the number of rules r does not depend on n, the total time required for one iteration of the T_{P} operator is also $O((\exp_{k-1}(n^{t+1}))^2)$.

From the above we get that in order to produce the minimum Herbrand model M_{P} of P, we need time $O(\exp_{k-1}(n^{t+1})) \cdot O((\exp_{k-1}(n^{t+1}))^2) = O((\exp_{k-1}(n^{t+1}))^3)$. By the properties of the function exp, it is $(\exp_{k-1}(n^{t+1}))^3 \leq \exp_{k-1}(3n^{t+1}) \leq \exp_{k-1}(n^{t+2})$.

We conclude that the running time of the Turing machine is $O(\exp_{k-1}(n^q))$ for $q = t+2$. The Turing machine returns *yes* if and only if `accept` is true in the minimum Herbrand model M_{P}. This completes the proof of the lemma. □

References

Dantsin, E., Eiter, T., Gottlob, 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.

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

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.