Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

Characterization of the Expressivity of Existential Rule Queries

Sebastian Rudolph and **Michaël Thomazo** *
Technische Universität Dresden, Germany
{sebastian.rudolph,michael.thomazo}@tu-dresden.de

Abstract

Existential rules (also known as Datalog$^{\pm}$ or tuple-generating dependencies) have been intensively studied in recent years as a prominent formalism in knowledge representation and database systems. We consider them here as a querying formalism, extending classical Datalog, the language of deductive databases. It is well known that the classes of databases recognized by (Boolean) existential rule queries are closed under homomorphisms. Also, due to the existence of a semi-decision procedure (the chase), these database classes are recursively enumerable. We show that, conversely, every homomorphism-closed recursively enumerable query can be expressed as an existential rule query, thus arriving at a precise characterization of existential rules by model-theoretic and computational properties. Although the result is very intuitive, the proof turns out to be non-trivial. This result can be seen as a very expressive counterpart of the prominent Lyndon-Łos-Tarski-Theorem characterizing the homomorphism-closed fragment of first-order logic. Notably, our result does not presume the existence of any additional built-in structure on the queried data, such as a linear order on the domain, which is a typical requirement for other characterizations in the spirit of descriptive complexity.

1 Introduction

The field of logic-based knowledge representation comprises a great variety of formalisms for specifying and querying knowledge. Given the tradeoff between declarative expressivity on the one hand and computational cost on the other hand, there is no unique formalism universally deployable in all of the numerous different usage scenarios. Thus, the available formalisms need to be categorized along the dimensions of expressive power and hardness of computation. For both aspects, the categorization can be *relative* or *absolute*.

Considering *relative expressivity* means to ask if every sentence or query[1] of one logic formalism can be equivalently expressed in the other. Investigating *relative computational properties* means to determine if the satisfaction problem in one formalism can be translated into the satisfaction problem in the other formalism using appropriate reductions (such as many-to-one or Turing reductions).

While these comparative investigations allow to relate and rank the considered formalisms, *absolute* characterizations typically provide much more profound insights into the true nature of logical languages. Absolute results relate logical formalisms to external measures of expressivity and computation. Among other benefits, such findings help establishing non-expressibility and non-reducibility results which are hard to obtain otherwise.

An absolute *computational* characterization of a logical formalism is achieved by determining the complexity or decidability or recursive enumerability of the corresponding satisfaction problem. Proving such computational properties of a given formalism is often a standard exercise. Contrariwise, showing that a logical formalism is capable of expressing *every* query exhibiting certain computational properties is usually a nontrivial undertaking and constitutes the subject of the field of *descriptive complexity theory* [Immerman, 1999]. As an example result from that line of research, it was established that first-order logic sentences exactly correspond to the properties of interpretations verifiable in AC^0, i.e., they can be checked by polynomial-size Boolean circuits of bounded depth.

One way of absolutely categorizing *expressivity* aspects of logical languages in a syntax-independent way, is via model-theoretic considerations. Intuitively, a formalism is more expressive than another if it allows for distinguishing two interpretations which are indistinguishable by the other formalism. Indistinguishability often can be characterized by the set of models of a certain type of queries being closed under certain operations. Examples for this are manifold: closure under intersection for Horn logics, bisimulation invariance in modal logics, closure under disjoint union of first-order sentences where no universal quantifier occurs inside the scope of an existential quantifier, and many more (see,

*Research supported by the Alexander von Humboldt Foundation.

[1] Since we are only concerned with satisfaction, we do not distinguish between the notions of a logical sentence and a (Boolean) query in this paper. All queries dealt with in this paper are assumed to be Boolean.

e.g., [Chang and Keisler, 1973] for more examples). A very natural such property is closure under homomorphism. It particularly holds for query languages that are supposed to check if substructures of a certain shape exist in an interpretation or database, a prominent example being Datalog and its various fragments (cf. [Rudolph and Krötzsch, 2013]). While it is typically not difficult to establish that such a model class closure property holds for a logical formalism, showing a converse property tends to be much more intricate. Next to a few others, a classical example of such a non-trivial result is the Lyndon-Łos-Tarski-Theorem stating that the set of models of a first-order sentence is homomorphism-closed if and only if it can be expressed in positive existential first-order logic, i.e., without using negation or universal quantification.

Inspecting this last result, we find that the obtained characterization of positive existential first-order logic is still somewhat relative since it refers to general first-order logic. However, with the aforementioned descriptive complexity result in place, one could combine the model-theoretic and the computational perspective to arrive at a characterization not referring to any other logical formalism: The class of queries expressible in positive existential first-order logic coincides with the class of queries that can be evaluated in AC^0 and whose set of models is closed under homomorphisms.

After making clear the general motivation and the thrust of our investigation by means of this low-level example, we now set out to achieve an absolute characterization for a very expressive formalism which has drawn a lot of attention in the last years: existential rules, which are known under a variety of other names (tuple-generating dependencies [Abiteboul *et al.*, 1994], Datalog$^{\pm}$ [Calì *et al.*, 2013], conceptual graph rules [Mugnier, 2009]). Their original use was to impose integrity constraints on a database, but they have been more recently used as a modeling language for ontologies. This formalism can also be seen, as in this paper, as a query language, as it was originally the case for its parent, Datalog, the language of deductive databases.

Surprisingly enough, an absolute characterization of plain existential rule queries has not been attempted so far. Trivial upper bounds can be proposed: First, the well-known *chase* procedure constitutes a semi-decision procedure for answering existential rule queries. Consequently, the set of finite relational structures (also referred to as databases) satisfying an existential rule query must be recursively enumerable.[2] Second, it is well-known and easy to show that this set is also closed under homomorphisms. The central contribution of this paper is to show that these two conditions together are in fact tight: any query Q, where the set of databases satisfying Q is both recursively enumerable and closed under homomorphisms, is equivalent to an existential rule query. Thereby, we arrive at the wanted characterization:

> *A query is expressible with existential rules iff its set of satisfying databases is recursively enumerable and homomorphism-closed.*

[2] More precisely, we should say recursively enumerable up to isomorphism. To avoid these technicalities, we assume the individuals of databases to come from a countably infinite reservoir of standard names.

While this result fits very well with intuition and may seem rather straightforward, establishing the "if" part is not at all trivial. To prove it, we simulate the computation of a Turing machine recognizing a query with the given properties.[3] Simulating a Turing machine given a correctly represented tape is classical with existential rules [Baget *et al.*, 2011a]. However, creating the representation of a tape from a database only through the use of existential rules requires some work. Indeed, existing techniques to create such a tape heavily rely on the use of two ingredients: a linear order on the elements of the domain, and a restricted form of negation. The first is in particular used to enumerate tuples, while the second is used to check the absence of facts. The most prominent use of these ingredients may be the capturing result of PTIME queries by semi-positive Datalog on linearly ordered databases [Abiteboul *et al.*, 1994].

Instead of considering a linear order, the existential rule query that we define for a given Turing machine will generate all finite lists containing elements of the domain. Some of these lists do not correspond to a linear order of the domain elements. We will present in Section 3 how to create a Turing machine tape from a database and such a list. An important question that needs to be tackled is the following: is it problematic if a Turing machine accepts when the tape has been created based on an enumeration that was not a linear order? Answering this question negatively is the topic of Section 4.

Last, we will need, given an enumeration of the terms of a database, to generate the corresponding tape. Without input negation, it is not possible to create a unique tape containing exactly the information corresponding to the actual database. Instead, we generate all possible databases on a given vocabulary. From all those "database candidates", we single out those databases inconsistent with the initial data. We then run the Turing machine on the tape corresponding to each candidate. We show that if each candidate is either inconsistent with the original data or leads to an accepting state of the Turing machine, then the query recognizes the structure parameterized by the enumeration. This is the topic of Section 5. Proofs not presented here are available at: https://ddll.inf.tu-dresden.de/web/Techreport3019/en.

2 Preliminaries

We assume the reader to be familiar with Turing machines (see [Papadimitriou, 1994] or [Arora and Barak, 2009]). A language is *recursively enumerable* if there is a Turing machine that accepts on any word of the language and does not terminate on any word that does not belong to the language. We are interested in decision problems on databases: the *encoding* of the database on a Turing machine tape is of importance. We describe the considered encoding in Section 3.

We consider two countable disjoint sets V and Δ of *variables* and *domain elements*, respectively. Elements of $V \cup \Delta$ are also called *terms*. We consider two finite disjoint sets $\mathcal{P}_i$

[3] As made more formal later, it is convenient to "semantically identify" a query with the set of databases satisfying it. This justifies to speak of a query itself being recognized by a Turing machine or closed under homomorphisms or recursively enumerable. For the sake of brevity, we will make extensive use of such wordings.

and $\mathcal{P}_e$ of *intensional predicates* and *extensional predicates*. Each predicate is either intensional or extensional and possesses an *arity* $n \in \mathbb{N}$. We assume w.l.o.g. that all extensional predicates have the same arity k. An *atom* is an expression a of the form $p(x_1, \ldots, x_n)$ where p is a predicate of arity n and $x_1, \ldots, x_n$ are terms. The terms of a are denoted by $\mathit{terms}(a)$. The terms of a set of atoms A are defined by $\cup_{a \in A} \mathit{terms}(a)$. Given two sets of atoms A and B, a *homomorphism* from A to B is a mapping π from $\mathit{terms}(A)$ to $\mathit{terms}(B)$ such that if $p(x_1, \ldots, x_n) \in A$, then $p(\pi(x_1), \ldots, \pi(x_n)) \in B$. An *isomorphism* from A to B is a bijective homomorphism π from A to B for which π^{-1} is also a homomorphism. A *database* (on some set $\mathcal{P}$ of predicates) is a finite set D of atoms with terms from Δ and predicates from $\mathcal{P}$. We assume (w.l.o.g.) that there exists a predicate $p \in \mathcal{P}_e$, denoted by ACDom such that $\mathit{ACDom}(x)$ holds for every term $x \in \mathit{terms}(D)$. Given a set of extensional predicates $\mathcal{P}_e$, a *(Boolean) query* is a subset of the databases on $\mathcal{P}_e$ that is closed under isomorphism.[4] A query q is said to be *closed under homomorphism* if for all $D_1 \in q$, if there is a homomorphism from D_1 to D_2, then $D_2 \in q$. An *existential rule* is a first-order formula of the form

$$\forall \bar{x}\ \forall \bar{y}\ B[\bar{x}, \bar{y}] \rightarrow \exists \bar{z}\ H[\bar{y}, \bar{z}],$$

where $\bar{x}, \bar{y}$ and $\bar{z}$ are tuples of variables, B is a conjunction of atoms (of intensional or extensional predicates) such that $\mathit{terms}(B) = \{\bar{x}, \bar{y}\}$ and H is a conjunction of atoms (of intensional predicates) such that $\mathit{terms}(H) = \{\bar{y}, \bar{z}\}$. A rule $\forall \bar{x}\ \forall \bar{y}\ B[\bar{x}, \bar{y}] \rightarrow \exists \bar{z}\ H[\bar{y}, \bar{z}]$ is *applicable* to a database D if there is a homomorphism from B to D. The result of this application is a new database $D \cup H'$, where H' is equal to H with each variable replaced by its image under π if defined, and by a some "new element" from $\Delta \setminus \mathit{terms}(D)$. We now briefly introduce the *chase* [Maier *et al.*, 1979; Beeri and Vardi, 1984]. Given a set of existential rules $\mathcal{R}$, a breadth-first application of applicable rules generates a potentially infinite sequence of databases. Their union is uniquely defined (up to homomorphic equivalence) and is called the $\mathcal{R}$-*chase* of D with respect to $\mathcal{R}$. We call the domain elements present in the chase but not in the original database (i.e., those added by some rule application) *fresh elements*.

An *existential rule query* $q_{\mathcal{R}}$ is a query represented by a set $\mathcal{R}$ of existential rules with a special predicate **goal**. A database D belongs to this query if **goal** belongs to the chase of D with respect to $\mathcal{R}$. It is clear that existential rule queries are closed under homomorphisms. Moreover, as constructing the chase and continuously checking for containedness of **goal** is a semi-decision procedure for $D \in q_{\mathcal{R}}$, these queries are also recursively enumerable.

For the sake of brevity, we will from now on omit quantifiers from existential rules, adopting the following convention: all variables occurring in the body B are universally quantified, all others existentially quantified.

[4]This definition reflects the common understanding of a query that it "[...] should be independent of the representation of the data in a data base and should treat the elements of the data base as uninterpreted objects" [Chandra and Harel, 1980]. This understanding also justifies why we do not distinguish the domain elements into constants and labeled nulls, as it sometimes done in the literature, and why we do not allow for constants in existential rules.

3 Turing Machine: Tape Representation

In this section, we describe the tape representation used to simulate a Turing machine with existential rules. We split the presentation in two: first, we describe how to transform a database (i.e., a set of facts, endowed with some additional structure) into a linearized tape representation. Second, we explain how to represent such a tape as a relational structure and how to simulate the Turing machine using existential rules.

3.1 Tape Representation of a Database

To represent deterministically a database on a tape, we make use of a linear order on its terms. However, since we do not have access to a linear order, we present a more general transformation associating a database and a sequence (potentially with repetitions) of its terms with a tape. Let us thus consider a database D and ℓ a sequence of its domain elements. The elements of ℓ are denoted by the binary representation of their rank in the sequence. Let us remark that this implies that a single domain individual of the database may have several representations, as seen in Example 1.

Definition 1 (D-list) *Let D be a database. A D-list is a sequence (possibly with repetitions) of terms of D, denoted by $(t_1, \ldots, t_n)$. A* representation *of a term t of D appearing in a D-list ℓ is a binary representation of a rank of t in ℓ.*

Since a term may appear several times in a D-list, it may thus have several representations.

Example 1 (Representations of an individual) *Let $\{a, b\}$ be the domain elements of a database D. A D-list is (a, b, a, a). Thus, a has three representations, $00, 10$ and 11, while b has one representation 01.*

A D-list naturally induces a linear order on the binary representations of its elements. We next describe how, relying on ℓ, we can come up with a tape representation of D. Remember that we assume w.l.o.g. that all our database predicates have a uniform arity of k. The linear order on representations in ℓ induces a linear order on k-tuples of representations in ℓ which we use for constructing our tape: after a "beginning of tape" symbol ($\#$), we start from the first k-tuple according to the mentioned order, write some information about it, and proceed to the next tuple until the last tuple is treated. For each tuple $(r_1, \ldots, r_k)$ of representations, we write for each predicate (in lexicographic order) p if $p(t_1, \ldots, t_k)$ holds in D or not, where where $t_1, \ldots, t_k$ are the domain elements represented by $r_1, \ldots, r_k$, respectively.

Example 2 *Let us consider a database D_e having as domain $\{a, b\}$ and two facts: $r(a, b)$ and $p(a, a)$. We consider the following D-list, which is a linear order : (a, b). The representation is the following:*

$$\#p1r0p0r1p0r0p0r0$$

The first $p1$ means that $p(a, a)$ holds. The first $r0$ means that $r(a, a)$ does not hold. The rest is interpreted similarly.

Given a database D and a D-list ℓ, we will denote by $\mathcal{T}(D, \ell)$ the tape representation that we described so far.

Last, we already pointed out that Turing machines work on strings that are representations of the input. More specifically, by a recursively enumerable query q, we mean a query for which there exists a Turing machine $\mathbb{M}_q$ which recognizes the following language:

$\{\mathcal{T}(D,\ell) \mid D$ is contained in q and
ℓ is a linear order on the elements of $D\}$.

3.2 Encoding a Tape in a Database

We now describe how a tape $\mathcal{T}(D,\ell)$ is represented by means of database atoms.

Definition 2 (Relational representation of $\mathcal{T}(D,\ell)$) *Let D be a database and let ℓ be a D-list. Let n be the number of extensional predicates and t the number of tuples. The relational representation of the tape $\mathcal{T}(D,\ell)$ is given by the following atoms over some domain individuals db (representing the database itself) and $cell_0, \ldots, cell_{2nt}$ (representing the tape cells) using the predicates* begin *(binary, associating the database-representing individual with the first tape cell),* next *(binary, associating each tape cell with the subsequent one), as well as, for every symbol σ that might occur on the tape,* symbol_σ *(unary, used to assign to each cell its content):*

- $\textit{begin}(tape, cell)$
- $\textit{symbol}_{\#}(cell_0)$
- $\textit{next}(cell_i, cell_{i+1})$ *for every* $i < 2n(t+1)$
- $\textit{symbol}_p(cell_{2nj+2i})$ *for p being the ith predicate and* $j \leq t$
- $\textit{symbol}_1(cell_{2nj+2i+1})$ *if* $p(t_1,\ldots,t_k)$ *holds for the ith predicate p and jth tuple* $(r_1,\ldots,r_k)$ *representing* $(t_1,\ldots,t_k)$
- $\textit{symbol}_0(cell_{2nj+2i+1})$ *if not* $\textit{symbol}_1(cell_{2nj+2i+1})$.

Given such a representation of $\mathcal{T}(D,\ell)$ and an according representative x_ℓ of ℓ, crafting a set of existential rules $\mathcal{R}_{\mathbb{M}_q}$ that simulate a given Turing $\mathbb{M}_q$ machine on that tape and derive $\textit{rec}(db, x_e ll)$ exactly if $\mathbb{M}_q$ accepts $\mathcal{T}(D,\ell)$ is common knowledge. The interested reader can consult [Baget *et al.*, 2011a] for this.

4 Correct and Incorrect Orderings

In the previous section, we presented the tape representation of a pair (D,ℓ), where D is a database and ℓ a D-list. We now present how to create D-lists through existential rules. We make use of the predicate *ACDom* that holds for each individual of the database.

Definition 3 (List annotator) *The* list annotator*, denoted by $\mathcal{R}_a$ is the following set of rules:*

- $\textit{ACDom}(x) \rightarrow \textit{link}(x,y) \wedge \textit{first}(y) \wedge \textit{last}(y)$
- $\textit{ACDom}(x) \rightarrow \textit{link}(x,y) \wedge \textit{first}(y) \wedge \textit{partial}(y)$
- $\textit{ACDom}(x) \wedge \textit{partial}(y) \rightarrow \textit{succ}(y,z) \wedge \textit{link}(x,z) \wedge \textit{partial}(z)$
- $\textit{ACDom}(x) \wedge \textit{partial}(y) \rightarrow \textit{succ}(y,z) \wedge \textit{link}(x,z) \wedge \textit{last}(z)$.

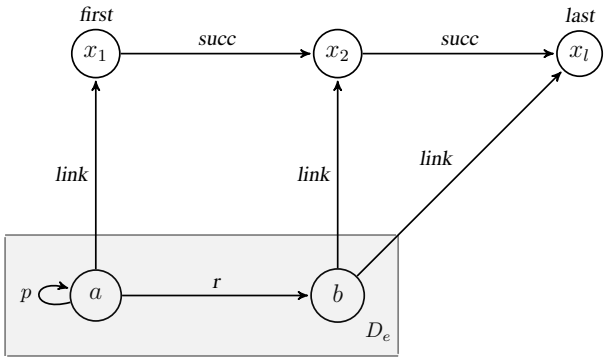

Figure 1: Partial effect of the D-annotator on Example 3

Intuitively, the list annotator makes, for every D-list ℓ, the representations of ℓ available as additional domain elements in our database, defines a unary predicate *first* to mark the first element of ℓ, a unary predicate *last* to denote the last, a binary predicate *succ* which connects a representation element with its immediate successor and a binary predicate *link* which links a representation element back to the original domain element it represents in ℓ. Note that, with these predicates in place, it is classical (see for instance [Abiteboul *et al.*, 1994]) to specify Datalog rules that define a linear order on the set of all k-tuples of representation elements, (where k is the maximum arity of a predicate in the original database).

Definition 4 (Representative of a D-list) *Let D be a database, and $\ell = (a_1,\ldots,a_n)$ be a D-list. Assume there are fresh elements $x_1,\ldots,x_n$ such that:*

- $\textit{first}(x_1)$ *and* $\textit{last}(x_n)$ *hold in D;*
- *for all i such that* $1 \leq i < n$, $\textit{succ}(x_i, x_{i+1})$ *holds in D;*
- *for all i such that* $1 \leq i \leq n$, $\textit{link}(a_i, x_i)$ *holds in D;*
- *no other atom of predicates* last, succ, link *has some x_i as argument.*

Then x_n is called a representative *of ℓ.*

Example 3 (ℓ-annotation) *Figure 1 shows part of the structure that is created by the list annotator on the database $p(a,a) \wedge r(a,b)$. x_ℓ is a representative of the D-list (a,b,b).*

The list annotations that interest us are the ones that correspond to some linear order. However, given the list annotator introduced, many more will be generated: domain elements might be left out or referenced multiple times. This raises the following question: if q is a homomorphism-closed query, $\mathbb{M}_q$ is a Turing machine for q, is it possible that $\mathbb{M}_q$ accepts $\mathcal{T}(D,\ell)$ for some pair (D,ℓ), while D does not belong to q? In other words: may the creation of annotations that do not correspond to proper linear orders lead to "false positives"? Luckily, we can answer this question negatively, where the intuitive argument is the following: with each pair (D,ℓ), we associate a pair (D',ℓ') such that $\mathcal{T}(D,\ell) = \mathcal{T}(D',\ell')$, with the additional condition that ℓ' is a linear order on a subset of the domain elements of D'. Therefore, $\mathbb{M}_q$ accepting $\mathcal{T}(D',\ell')$ proves that D' belongs to q. Then by showing that

there is a homomorphism from D' to D allows to conclude that D, in fact, belongs to q.

Definition 5 (Singularized Database) *Let D be a database instance, let ℓ be a D-list. The* singularization $\mathit{sing}(D, \ell) = (D', \ell')$ *of D with respect to ℓ is defined as follows:*

- *ℓ' has the same length as ℓ, and the i^{th} element of ℓ' is the i^{th} element of ℓ with an additional i superscript,*
- *$p(x_1^{i_1}, \ldots, x_k^{i_k}) \in D'$ iff $p(x_1, \ldots, x_k) \in D$.*

Example 4 *Let us consider the database D_e of Example 2 with a D-list ℓ being (a, b, b). The singularization of D_e with respect to ℓ is (D'_e, ℓ') with $D'_e = \{r(a^1, b^2), r(a^1, b^3), p(a^1, a^1)\}$ and $\ell' = (a^1, b^2, b^3)$.*

Proposition 1 *For any database D and any D-list ℓ, $\mathcal{T}(D, \ell) = \mathcal{T}(\mathit{sing}(D, \ell))$.*

Proof: We put $\ell = (x_1, \ldots, x_p)$ and $\ell' = (x'_1, \ldots, x'_p)$. The binary representation of x_i and x'_i are equal. By construction of $\mathit{sing}(D, \ell)$, the every predicate that holds for the tuple $(x'_1, \ldots, x'_k)$ holds as well for the tuple $(x_1, \ldots, x_k)$. For each tuple, the same word is thus written on the tape, and exactly the same tuples are considered. □

Proposition 2 *Let D be a database, let ℓ be a D-list, and $(D', \ell') = \mathit{sing}(D, \ell)$. There is a homomorphism from D' to D.*

Proof: Let us consider π, that associates the i^{th} term of ℓ' with the i^{th} term of ℓ. This function is well-defined, since all terms of ℓ' are distinct. π is a homomorphism from D to D': indeed, $p(x_1^{i_1}, \ldots, x_k^{i_k})$ belongs to D' if and only if $p(x_1, \ldots, x_k)$ belongs to D. □

Proposition 3 *Let q be a query closed under homomorphism. Let $\mathbb{M}_q$ be a Turing machine recognizing q. If $\mathbb{M}_q$ recognizes the tape representation of (D, ℓ) with ℓ being an arbitrary D-list, then D belongs to q.*

Proof: Let us consider the singularization of D with respect to ℓ. By Proposition 1, the tape representing $\mathit{sing}(D, \ell) = (D', \ell')$ is the same as the tape representing (D, ℓ). Thus, $\mathbb{M}_q$ accepts on that tape. Since ℓ is a linear order on the terms of D', by definition of the Turing machine, D' belongs to q. Since q is closed under homomorphisms, and by Proposition 2, D belongs as well to q. □

5 Database Completion

5.1 General Description

We now describe how to initialize the tape of the Turing machine. The first step is, given a D-list ℓ, to generate all the possible databases on $\mathcal{P}_e$ with terms from ℓ. Moreover, we want to do this in such a way that through existential rules, one can decide if a given atom is present or not in the generated database. The second step is to write the generated databases on tapes (which are parameterized by the D-list under consideration and the generated database).

To generate all possible databases, we enumerate tuples according to the D-list, making every possible choice regarding the validity of atoms referring to the given tuple. This choice is represented by fresh intensional predicates, $\Omega = \{\omega_P \mid P \subseteq \mathcal{P}_e\}$. These predicates have arity $k + 2$. The first k positions are filled with the tuple under consideration. The $k + 1^{\text{th}}$ position is filled by a fresh element representing the portion of the database that has been generated so far, while the last position is used to remember (the representative of) the D-list currently used. Intuitively, $\omega_P(x_1, \ldots, x_k, x, x_\ell)$ holds if and only if, for any $p \in \mathcal{P}_e$, $p(y_1, \ldots, y_k)$ holds in the partial database represented by x iff $p \in P$, where y_i is the unique term such that $\mathit{link}_\ell(y_i, x_i)$.

Once these databases have been generated, we create the corresponding tapes and simulate the Turing machine $\mathbb{M}_q$ on each of those tapes. If a generated database contains (at least) all the atoms of the original D, the simulation accepts. Otherwise, the simulation may not terminate, but we shortcut this case by "exceptionally accepting" all generated databases where a we detect discrepancies to the original database.

5.2 Formalization of the Construction

Generation of the databases We finally introduce the rules for the part $\mathcal{R}_c$ of the query, dealing with database generation and tape creation. For each tuple, we consider every possible choice regarding which predicates hold for this tuple. We create $|\Omega|$ rules for the first tuple:

$$\mathit{first}_k(x_1, \ldots, x_k, x_\ell) \rightarrow \omega(x_1, \ldots, x_k, x, x_\ell), \omega \in \Omega. \quad (1)$$

For each application of such a rule, the fresh element created by the instantiation of x represents a database where the atoms for the first tuple are fully specified, but no other tuple is known. To make choices for the other tuples as well, one uses the successor relation on tuples built from the D-list. We allow for any possible choice, thus creating $|\Omega|^2$ rules.

$$\begin{aligned}\omega(x_1, \ldots, x_k, x, x_\ell)& \\ \wedge\ \mathit{succ}_k(x_1, \ldots, x_k, x_{k+1}, \ldots, x_{2k}, x_\ell)& \\ \rightarrow \mathit{step}(x, y, x_\ell) \wedge \omega'(x_{k+1}, \ldots, x_{2k}, y, x_\ell)& \\ \text{for all } \omega, \omega' \in \Omega. \quad (2)&\end{aligned}$$

To ease the construction of the tape, we propagate the "Ω-choices" already made for earlier tuples from one partial representation to the next:

$$\omega(x_1, \ldots, x_k, x, x_\ell) \wedge \mathit{step}(x, y, x_\ell) \rightarrow \omega(x_1, \ldots, x_k, y, x_\ell). \quad (3)$$

Of course, such "blind" choices may lead to inconsistencies with the present facts: we may choose that $p(a, b)$ does not hold, whereas it is stated in the original data. We check such inconsistencies (and trigger an "exceptional accept" as explained before) as follows:

$$\begin{aligned}\omega_P(x_1, \ldots, x_k, x, x_\ell) \wedge \bigwedge_{i=1}^{k} \mathit{link}_\ell(y_i, x_i) \wedge p(y_1, \ldots, y_k)& \\ \rightarrow \mathit{allaccept}(x, x_\ell) \text{ if } p \notin P. \quad (4)&\end{aligned}$$

When reaching $\mathit{last}_k(x_1, \ldots, x_k, x_\ell)$, a choice has been made for every predicate and every tuple, thus we have obtained a fully described database D_{x,x_ℓ} over the representations of elements in ℓ, which is represented by the fresh element x. It remains to verify that this fully specified database belongs to the query. To this end, we create a tape representing this database, and run the Turing machine $\mathbb{M}_q$ on it. Before describing how to perform these tasks, let us assume that if D_{x,x_ℓ} belongs to the query, then $\mathit{rec}(x, x_\ell)$ is derived. We propagate this information to the representatives of ever more partial databases, starting from the complete ones (i.e., those where the last_k predicate holds for the tuple treated last:

$$\omega(x_1, \ldots, x_k, x, x_\ell) \wedge \mathit{last}_k(x_1, \ldots, x_k, x_\ell) \rightarrow \mathit{complete}(x, x_\ell); \quad (5)$$

$$\mathit{complete}(x, x_\ell) \wedge \mathit{rec}(x, x_\ell) \rightarrow \mathit{allaccept}(x, x_\ell). \quad (6)$$

The acceptance information (irrespective if exceptional or regular) is then propagated through the tree of partial database representations:

$$\bigwedge_{\omega \in \Omega} \big(\mathit{step}(x, y_\omega, x_\ell) \wedge \omega(x_1, \ldots, x_k, y_\omega, x_\ell) \wedge \mathit{allaccept}(y_\omega, x_\ell)\big) \rightarrow \mathit{allaccept}(x, x_\ell). \quad (7)$$

If *allaccept* has been propagated up to the root, each possible way of completing the data associated with terms of ℓ is either inconsistent with D or is finally encoded into a tape accepted by $\mathbb{M}_q$. We mark then ℓ as an accepting enumeration.

$$\mathit{first}_k(x_1, \ldots, x_k, x_\ell) \wedge \bigwedge_{\omega \in \Omega} \omega(x_1, \ldots, x_k, y_\omega, x_\ell) \wedge \mathit{allaccept}(y_\omega, x_\ell) \rightarrow \mathit{accenum}(x_\ell). \quad (8)$$

Creation of the tape We last present, for each x representing a database D_{x,x_ℓ} the creation of the initial tape on which the Turing machine is run. This is done as follows: we enumerate the tuples in the order induced by ℓ. For a given tuple, $(x_1, \ldots, x_k)$, there is exactly one atom of the form $\omega_P(x_1, \ldots, x_k, x, x_\ell)$. We thus add at the end of the tape built so far all the information regarding this tuple. We first initialize by creating the first cell.

$$\mathit{complete}(x, x_\ell) \rightarrow \mathit{begin}(x, y) \wedge \mathit{symbol}_\#(y) \wedge \mathit{needed}(y, \bar{x}, x, x_\ell) \wedge \mathit{first}_k(\bar{x}, x_\ell). \quad (9)$$

The *needed* predicate indicates which tuple should be written to the right of the current cell. This operation is done thanks to the following rules:

$$\mathit{needed}(c_0, \bar{x}, x, x_\ell) \wedge \omega_P(\bar{x}, x, x_\ell) \wedge \mathit{succ}_k(\bar{x}, \bar{y}, x_\ell) \rightarrow \bigwedge_{i=0}^{2n-1} \mathit{next}(c_i, c_{i+1}) \wedge \bigwedge_{i=1}^{n} \mathit{symbol}_{p_i}(c_{2i-1}) \wedge \mathit{symbol}_{\delta_{p_i,P}}(c_{2i}) \wedge \mathit{needed}(c_{2i}, \bar{y}, x, x_\ell), \quad (10)$$

where there is such a rule for each $P \subseteq \mathcal{P}_e$, and $\delta_{p_i,P}$ denotes 1 if $p_i \in P$ and 0 otherwise. In English, this rule states that if at the cell c_0 of the tape on which we write the representation of D_{x,x_ℓ} the information concerning the tuple $\bar{x}$ is required, if ω_P describes this information, and if $\bar{y}$ is the next tuple in lexicographic order, then we create $2n$ new cells with the relevant information and we declare that the information regarding $\bar{y}$ is needed at its right.

5.3 Sketch of Proof of the Construction

We now consider the query $q_\mathcal{R}$ with $\mathcal{R} = \mathcal{R}_a \cup \mathcal{R}_c \cup \mathcal{R}_{\mathbb{M}_q}$. Let D be a database, and ℓ be a D-list. We first prove that for any guessed database D' on the terms of $\mathit{sing}(D, \ell)$, there is a fresh element $x_{D'}$ representing D' (Proposition 4). Then we show that the rules create the representation of the tape associated with (D', ℓ) when applied from $x_{D'}$ (Proposition 5). Finally, assuming that $\mathit{rec}(x_{D'}, x_\ell)$ is derived whenever $\mathcal{T}(D', \ell)$ is accepted by $\mathbb{M}_q$, we show that $\mathit{accenum}(x_\ell)$ is derived whenever $\mathcal{T}(D, \ell)$ is accepted by $\mathbb{M}_q$ (Proposition 6).

Definition 6 (Alternative) *Let D be a database, ℓ be a D-list. An* alternative *D' for (D, ℓ) is a database on the terms of $\mathit{sing}(D, \ell)$. A* representation *of D' is a set of atoms of the form $\omega_P(\bar{t}, x, x_\ell)$, where x is a free variable, x_ℓ a representative of ℓ, and ω_P is such that $p(\bar{t}) \in D'$ if and only if $p \in P$.*

Proposition 4 *Let D be a database, ℓ a D-list. Let D' be an alternative for (D, ℓ). There exists $x_{D'}$ in the $(\mathcal{R}_a \cup \mathcal{R}_c)$-chase of D such that the set of atoms of the form $\omega_P(\bar{t}, x_{D'}, \ell)$ in the chase is a representation of D'.*

A fresh element $x_{D'}$ as described in the previous property is then called a *representative* of D'.

Proposition 5 *Let D be a database, let ℓ be a D-list. Let D' be an alternative for (D, ℓ), $x_{D'}$ a representative of D'. There exists a sequence of fresh elements in the $(\mathcal{R}_a \cup \mathcal{R}_c)$-chase of D that form a representation of the tape $\mathcal{T}(D', \ell)$.*

Proposition 6 *Let D be a database, let ℓ be a D-list. $\mathcal{T}(D, \ell)$ is accepted by $\mathbb{M}_q$ if and only if there exists a representative x_ℓ of ℓ in the $(\mathcal{R}_{\mathbb{M}_q} \cup \mathcal{R}_a \cup \mathcal{R}_c)$-chase of D for which $\mathit{accenum}(x_\ell)$ holds.*

Proof (sketch): Notice that an alternative for (D, ℓ) is either detected by Rule (4) or there is a homomorphism from $\mathit{sing}(D, \ell)$ into it. Thus, if D belongs to a homomorphism-closed query, then $\mathit{rec}(x, x_\ell)$ holds for all representatives x of alternatives of (D, ℓ) in the canonical model, with x_ℓ being a representative of ℓ. We conclude by Rules (6) and (7). □

Last, by Proposition 3, we know that a database D belongs to q if and only if there exists a D-list ℓ such that $\mathcal{T}(D, \ell)$ is recognized by $\mathbb{M}_q$. This thus proves the following theorem.

Theorem 1 *Let q be a homomorphism-closed query, and $\mathbb{M}_q$ a Turing machine recognizing it. The existential rule query $q_\mathcal{R}$ with $\mathcal{R} = \mathcal{R}_{\mathbb{M}_q} \cup \mathcal{R}_a \cup \mathcal{R}_c \cup \{\mathit{accenum}(x_\ell) \rightarrow \mathbf{goal}\}$ is such that a database D on $\mathcal{P}_e$ belongs to q if and only if D belongs to $q_\mathcal{R}$.*

This in turn implies our main result: every homomorphism-closed recursively enumerable query is expressible as an existential rule query.

6 Discussion and Future Work

In this work, we have considered existential rule queries. Existential rules have been intensively studied in recent years as a prominent formalism in knowledge representation and databases. Quite surprisingly, the expressivity of this formalism when considered as a query language has not been studied so far. We provided a clear characterization of this expressive power by showing that there are no further limits beyond the obvious: existential rule queries are exactly those queries which are preserved under homomorphisms and for which a semi-decision procedure exists. The beauty of this result lies in the absence of additional requirements regarding the database (such as a linear order on the domain elements or the presence of complement predicates). Consequently the major hurdle to be overcome was to generate appropriate tape representations without relying on a predefined linear order nor on input negation. We showed that this can be achieved by a brute force approach of creating all enumerations of domain elements and for every such enumeration all corresponding full databases. The final trick was to organize these proliferating enumerations and guessed databases in a way that a query match is correctly detected, despite the existence of "fake" linear orders and the side-by-side existence of incoherent databases, overly filled databases, and correct databases. The assumption that the query is preserved under homomorphism had to be heavily exploited.

Besides its elegance, the established result can be useful for clarifying expressivity questions. It is now clear that *every* query for which homomorphism preservation and semi-decidability in any Turing-equivalent computing paradigm can be established (by whatever means) must be expressible as an existential rule query. Conversely we know that every query *not* expressible via existential rules must violate one of these two conditions. Moreover, since we have shown the formalism to be complete for the class of queries satisfying the two conditions, it does not make sense to look for more expressive extensions of existential rule queries which are homomorphism-closed. On a side note, since our proof is constructive, we have provided a generic way of turning a Turing machine formulation of the query into an existential rule query, although in most cases certainly a suboptimal one.

Our ongoing work is focused on finding similar capturing results for homomorphism-closed classes of queries on different complexity levels (P, NP, PSPACE, EXPTIME, and others) ideally linked to natural syntactic restrictions of existential rules. To this end, we may draw on prior work on complexities of different decidable fragments of existential rules [Baget *et al.*, 2011b; Krötzsch and Rudolph, 2011], exploit existing results linking certain complexities to existential rules classes extended by mild forms of negation [Abiteboul *et al.*, 1994; Gottlob *et al.*, 2014], and develop results in the spirit of [Feder and Vardi, 2003], showing that negation can be removed when considering only homomorphism preserved queries. However, results might not turn out as elegant as desired. For instance, the natural candidate for capturing the class of homomorphism-preserving polytime-computable queries, Datalog, has recently been shown to not fully capture that class [Dawar and Kreutzer, 2008].

References

[Abiteboul *et al.*, 1994] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1994.

[Arora and Barak, 2009] S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[Baget *et al.*, 2011a] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On Rules with Existential Variables: Walking the Decidability Line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.

[Baget *et al.*, 2011b] J.-F. Baget, M.-L. Mugnier, S. Rudolph, and M. Thomazo. Walking the complexity lines for generalized guarded existential rules. In *Proceedings of IJCAI'11*, pages 712–717, 2011.

[Beeri and Vardi, 1984] C. Beeri and M.Y. Vardi. A Proof Procedure for Data Dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[Calì *et al.*, 2013] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res. (JAIR)*, 48:115–174, 2013.

[Chandra and Harel, 1980] A. K. Chandra and D. Harel. Computable queries for relational data bases. *J. Comput. Syst. Sci.*, 21(2):156–178, 1980.

[Chang and Keisler, 1973] C. C. Chang and H. J. Keisler. *Model Theory*. Elsevier, 1973.

[Dawar and Kreutzer, 2008] A. Dawar and S. Kreutzer. On datalog vs. LFP. In *Proceedings of ICALP'08*, pages 160–171, 2008.

[Feder and Vardi, 2003] T. Feder and M. Y. Vardi. Homomorphism closed vs. existential positive. In *Proceedings of LICS'03*, pages 311–320, 2003.

[Gottlob *et al.*, 2014] G. Gottlob, S. Rudolph, and M. Simkus. Expressiveness of guarded existential rule languages. In *Proceedings of PODS'14*, pages 27–38, 2014.

[Immerman, 1999] N. Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.

[Krötzsch and Rudolph, 2011] M. Krötzsch and S. Rudolph. Extending decidable existential rules by joining acyclicity and guardedness. In *Proceedings of IJCAI'11*, pages 963–968, 2011.

[Maier *et al.*, 1979] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, 1979.

[Mugnier, 2009] M.-L. Mugnier. Conceptual graph rules and equivalent rules: A synthesis. In *Proceedings of ICCS'09*, pages 23–31, 2009.

[Papadimitriou, 1994] C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[Rudolph and Krötzsch, 2013] S. Rudolph and M. Krötzsch. Flag & check: Data access with monadically defined queries. In *Proceedings of PODS'13*, pages 151–162, 2013.