Tuple-Generating Dependencies Capture Complex Values

Maximilian Marx
Knowledge-Based Systems Group, TU Dresden, Dresden, Germany

Markus Krötzsch
Knowledge-Based Systems Group, TU Dresden, Dresden, Germany

Abstract

We formalise a variant of Datalog that allows complex values constructed by nesting elements of the input database in sets and tuples. We study its complexity and give a translation into sets of tuple-generating dependencies (TGDs) for which the standard chase terminates on any input database. We identify a fragment for which reasoning is tractable. As membership is undecidable for this fragment, we develop decidable sufficient conditions.

2012 ACM Subject Classification Theory of computation → Complexity theory and logic; Theory of computation → Constraint and logic programming; Theory of computation → Logic and databases

Keywords and phrases terminating standard chase, existential rules, Datalog, complexity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2022.13

Funding This work is partly supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) in project number 389792660 (TRR 248, Center for Perspicuous Systems), by the Bundesministerium für Bildung und Forschung (BMBF, Federal Ministry of Education and Research) in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), and by the Center for Advancing Electronics Dresden (cfaed).

Acknowledgements We would like to thank the anonymous reviewers for their detailed feedback that led to this revised version of the paper.

1 Introduction

Complex values (also called complex objects) are formed by combining domain elements into sets, tuples, and sets and tuples of other complex values. Numerous extensions of the relational data model with complex values have been studied since the 1980s: LDL1 [37, 6] and LPS [27] support non–nested sets, whereas COL [1] can express all complex values. Several fixed point logics for complex values, as well as their tractable fragments, have also been studied [33, 36, 22]. Extensions of Datalog include Relationlog [28], Set-extended Datalog [39], IQL [3], O-Logic [25] and Higher-order Datalog [14]. Abiteboul et al. [2] provide an extensive treatment of the complex values algebra and calculus, and briefly introduce a variant of stratified Datalog with complex values, though without any formal definition.

These expressive data models have recently regained much interest. A popular example are JSON objects, which can be viewed as sets of attribute-value pairs. Such "values" are at the heart of NoSQL systems, such as CouchDB and RethinkDB, and also supported by classical RDBMS, such as PostgreSQL and MariaDB. Query languages for JSON include J-Logic [23] and RNJL [9]. Another important example are rich graph models, such as Property Graph [34, 35] and the Wikidata knowledge graph [38], which are widely used in applications. There, one often deals with sets of "annotations" (e.g., attribute–value pairs) attached to edges, which can be naturally represented as complex values. Query languages such as MARPL [30], eMARPL [29], and G-CORE [5] have been proposed for this scenario.

Graphs also bring up the need for recursive queries, even for basic tasks such as reachability.

Languages like G-CORE go beyond this by supporting "paths as first class citizens" [5] in queries that return paths. Similar functionality can be realised using complex values:

▶ **Example 1.** Consider a database encoding a directed graph using facts $\mathsf{edge}(s,t)$ to denote edges from vertex s to vertex t. In our proposed formalism $\mathsf{Datalog}^{\mathrm{CV}}$, we can query for paths (represented as sets of edges, i.e., as sets of pairs of vertices) from x to y as follows:

$$\mathsf{edge}(x,y) \to \mathsf{path}(x,y,\{\langle x,y\rangle\}) \tag{1}$$

$$\mathsf{path}(x,y,P) \wedge \mathsf{edge}(y,z) \to \mathsf{path}(x,z,P \cup \{\langle y,z\rangle\}) \tag{2}$$

Intuitively, rule (1) states that whenever there is an edge from x to y, there is also a path going along that edge. Rule (2) extends a path from x to y along an edge from y to z.

Consider a database with the following facts:

$$\mathsf{edge}(a,b) \qquad \mathsf{edge}(b,c) \qquad \mathsf{edge}(a,c)$$
$$\mathsf{edge}(a,d) \qquad \mathsf{edge}(d,c) \qquad \mathsf{edge}(d,e)$$

Then we can derive the following three paths from a to c:

$$\mathsf{path}(a,c,\{\langle a,c\rangle\}) \qquad \mathsf{path}(a,c,\{\langle a,b\rangle,\langle b,c\rangle\}) \qquad \mathsf{path}(a,c,\{\langle a,d\rangle,\langle d,c\rangle\})$$

Other examples of even more complex recursive queries over complex values are found in symbolic AI, where Datalog with set values has been successfully used to reason with description logics and guarded Horn logics [32, 4, 13, 12].

Many of the above extensions involve declarative rules, which are of interest for applications such as data exchange or ontology-based query answering. Surprisingly, however, there is almost no practical support for such rules, even for limited uses of complex values. One of the most advanced systems available is *DLV-complex* [10], a logic programming engine that is not intended for database use (and only available as a 32bit-binary, i.e., limited to 4GB of memory). Hence, to realise set-based reasoning in practice, Carral et al. [13] have translated their "datalog plus sets" programs into sets of *tuple-generating dependencies* (TGDs), for which modern engines exist. Similar to Datalog, a bottom-up algorithm (the *standard chase*) can be used for reasoning, and Carral et al. show that, under certain restrictions on the order of rule applications, this approach will successfully terminate on any input database.

In this paper, we set out to generalise existing ad-hoc extensions into $\mathsf{Datalog}^{\mathrm{CV}}$, an extension of Datalog with full support for complex values. The language corresponds to the positive variant of Datalog with complex values as informally described by Abiteboul et al. [2], but we add a detailed analysis of complexity and expressivity, with a modern focus on tractable fragments and TGD-based implementation. Concretely, our contributions are as follows:

- we formalise Datalog with complex values ($\mathsf{Datalog}^{\mathrm{CV}}$) and obtain complexity bounds, showing that our formalism is intractable even for data complexity;
- we develop a translation into TGDs that preserves entailments in a certain sense, and for which the standard chase is guaranteed to terminate for all input databases and (in contrast to earlier work [13]) all strategies;
- we identify the tractable fragment (with respect to data complexity) of *bounded-cardinality programs*, which still supports non-trivial use of complex values, and develop two sufficient criteria for recognising it; and
- we show that, unlike for Datalog and TGDs, *linear rules* do not guarantee tractability.

2 Preliminaries

We consider fixed, pairwise disjoint, and countably infinite sets $\mathbf{C}$ of *constants*, $\mathbf{P}$ of *predicate names*, $\mathbf{V}$ of *variables*, and $\mathbf{N}$ of *labelled nulls*. With each predicate name $p \in \mathbf{P}$, we associate an *arity* $\mathsf{ar}(p) \in \mathbb{N}_{\geq 0}$. An *atom* is of the form $p(t_1, \ldots, t_\ell)$, where p is an ℓ-ary predicate name and $t_1, \ldots, t_\ell \in \mathbf{C} \cup \mathbf{V} \cup \mathbf{N}$ are *terms*. We may abbreviate such a list $t_1, \ldots, t_\ell$ as $\boldsymbol{t}$, and denote by $\boldsymbol{t}|_{i \mapsto s}$ the list $t_1, \ldots, t_{i-1}, s, t_{i+1}, \ldots, t_\ell$ obtained from $\boldsymbol{t}$ by replacing the i-th element by s. An atom is *null-free* if it does not contain any labelled nulls, *variable-free* if does not contain any variables, and *ground* if it is both null-free and variable-free. A *tuple-generating dependency* (TGD) or *rule* ρ is a formula of first-order logic of the form

$$\forall \boldsymbol{x}, \boldsymbol{y}.\ \varphi[\boldsymbol{x}, \boldsymbol{y}] \rightarrow \exists \boldsymbol{z}.\ \psi[\boldsymbol{y}, \boldsymbol{z}] \tag{3}$$

where (i) $\boldsymbol{x}, \boldsymbol{y}$ and $\boldsymbol{z}$ are mutually disjoint lists of variables, (ii) φ and ψ are conjunctions of null-free atoms, (iii) φ contains only variables from $\boldsymbol{x} \cup \boldsymbol{y}$, and (iv) ψ contains only variables from $\boldsymbol{x} \cup \boldsymbol{z}$.

We tacitly omit the universal quantifiers when writing TGDs, and call $\mathrm{body}(\rho) = \varphi$ the *body*, $\mathrm{head}(\rho) = \psi$ the *head*, and $\mathrm{frontier}(\rho) = \boldsymbol{y}$ the *frontier* of (3). We may treat conjunctions such as φ and ψ as sets of atoms, respectively.

A *boolean conjunctive query* (BCQ) q is a first-order sentence of the form $\exists \boldsymbol{x}.\ \varphi[x]$, where φ is a conjunction of null-free atoms. A *rule set* Σ is a finite set of TGDs. A predicate name p is an *intensional database predicate* (IDB) with respect to Σ if p occurs in the head of a rule in Σ. All other predicate names are *extensional database predicates* (EDB). An *EDB schema* is a finite set $\mathbf{P}^{\mathrm{EDB}} \subsetneq \mathbf{P}$ of predicate names. A rule set Σ is *compatible* with the EDB schema if all predicate names $p \in \mathbf{P}^{\mathrm{EDB}}$ are EDB predicates with respect to Σ. A *database instance* is a set of variable-free atoms. A *database* $\mathcal{D}$ over $\mathbf{P}^{\mathrm{EDB}}$ is a finite set of ground atoms using only predicate names from $\mathbf{P}^{\mathrm{EDB}}$. If $\mathcal{D}$ is used with rule set Σ, we always assume that $\mathcal{D}$ is over an EDB schema that Σ is compatible with.

Let $\mathcal{I}$ be a database instance, Σ a rule set, and $\mathcal{D}$ a database for Σ. Given a set A of atoms, a *homomorphism* is a function $h : A \rightarrow \mathcal{I}$ that maps terms in A to (variable-free) terms in $\mathcal{I}$, such that (i) $h(c) = c$ for all $c \in \mathbf{C}$ and (ii) $p(h(t_1), \ldots, h(t_\ell)) \in \mathcal{I}$ for all $p(t_1, \ldots, t_\ell) \in A$. A *match* of a rule ρ in $\mathcal{I}$ is a homomorphism $h : \mathrm{body}(\rho) \rightarrow \mathcal{I}$; it is *satisfied* in $\mathcal{I}$ if there is a homomorphism $h' : \mathrm{head}(\rho) \rightarrow \mathcal{I}$ with $h(x) = h'(x)$ for all $x \in \mathrm{frontier}(\rho)$. $\mathcal{I}$ *satisfies*

- a rule ρ (written $\mathcal{I} \models \rho$) if every match of ρ is satisfied,
- the rule set Σ (written $\mathcal{I} \models \Sigma$) if $\mathcal{I} \models \rho$ for all $\rho \in \Sigma$, and
- a BCQ $q = \exists \boldsymbol{x}.\ \varphi[\boldsymbol{x}]$ (written $\mathcal{I} \models q$) if there is a homomorphism $h : \varphi[\boldsymbol{x}] \rightarrow \mathcal{I}$.

$\mathcal{I}$ is a *model* of $\mathcal{D}$ and Σ (written $\mathcal{I} \models \mathcal{D}, \Sigma$) if $\mathcal{D} \subseteq \mathcal{I}$ and $\mathcal{I} \models \Sigma$. $\mathcal{D}$ and Σ *entail* a BCQ q (written $\Sigma, \mathcal{D} \models q$) if $\mathcal{I} \models q$ for all models $\mathcal{I}$ of $\mathcal{D}$ and Σ. A model $\mathcal{I} \models \mathcal{D}, \Sigma$ is *universal* if it admits homomorphisms into every model of Σ and $\mathcal{D}$. Universal models capture BCQ entailment: for q a BCQ and $\mathcal{I} \models \mathcal{D}, \Sigma$ a universal model, $\mathcal{I} \models q$ iff $\Sigma, \mathcal{D} \models q$.

Universal models can be computed by, e.g., the *standard chase* (or *restricted chase*) [7]. A *chase sequence* for a rule set Σ and database $\mathcal{D}$ is a sequence of database instances $\mathcal{D}_0, \mathcal{D}_1, \ldots$, such that: (i) $\mathcal{D}_0 = \mathcal{D}$; (ii) for every $i \geq 0$, there is a rule $\varphi[\boldsymbol{x}, \boldsymbol{y}] \rightarrow \exists \boldsymbol{z}.\ \psi[\boldsymbol{y}, \boldsymbol{z}]$ in Σ with an unsatisfied match h in $\mathcal{D}_i$, and $\mathcal{D}_{i+1} = \mathcal{D}_i \cup \psi[h'(\boldsymbol{y}), h'(\boldsymbol{z})]$, where h and h' agree on $\boldsymbol{y}$, and $h'(z) \in \mathbf{N}$ are distinct labelled nulls not occurring in $\mathcal{D}_i$ for all $z \in \boldsymbol{z}$; and (iii) if some rule ρ has a match h in some $\mathcal{D}_i$, then there is a $j > i$ such that h is satisfied in $\mathcal{D}_j$. The *chase* of $\mathcal{D}$ and Σ is the database instance $\bigcup_{i \geq 0} \mathcal{D}_i$. In general, termination of the chase can depend on the order of rule applications in step (ii) [17]; see [26] for a discussion.

3 Datalog with Complex Values

We now introduce $\mathsf{Datalog}^{\mathrm{CV}}$, an extension of Datalog in which values are terms built from tuples and sets of constants. We use sorts to represent different types of terms.

Syntax

The set $\mathcal{S}$ of *sorts* is defined inductively to contain (i) the *domain sort* Δ, (ii) for all $\tau \in \mathcal{S}$, the *set sort* $\{\tau\}$, and (iii) for all $\ell \geq 2$ and $\tau_1, \ldots, \tau_\ell \in \mathcal{S}$, the *tuple sort* $\langle \tau_1, \tau_2, \ldots, \tau_\ell \rangle$. A *subsort* of sort τ is any sort that occurs syntactically in τ, including τ itself. Every variable $v \in \mathbf{V}$ has a sort $\mathsf{sort}(v) \in \mathcal{S}$ such that the sets $\mathbf{V}_\tau = \{v \in \mathbf{V} \mid \mathsf{sort}(v) = \tau\}$ are countably infinite. The sets $\mathbf{T}_\tau$ of *terms of sort* τ are defined as follows:

1. $\mathbf{T}_\Delta \coloneqq \mathbf{C} \cup \mathbf{V}_\Delta$ (terms of the domain sort are constants or variables);
2. $\mathbf{T}_{\langle \tau_1, \ldots, \tau_\ell \rangle} \coloneqq \{\langle s_1, \ldots, s_\ell \rangle \mid s_i \in \mathbf{T}_{\tau_i} \text{ for } 1 \leq i \leq \ell\} \cup \mathbf{V}_{\langle \tau_1, \ldots, \tau_\ell \rangle}$ (terms of tuple sorts are tuples over the individual component sorts, or variables); and
3. $\mathbf{T}_{\{\tau\}} \coloneqq \{\{t_1, \ldots, t_n\} \mid n \geq 0, t_1, \ldots, t_n \in \mathbf{T}_\tau\} \cup \{(t_1 \cap t_2), (t_1 \cup t_2) \mid t_1, t_2 \in \mathbf{T}_{\{\tau\}}\} \cup \mathbf{V}_{\{\tau\}}$ (terms of set sorts are set literals over the component sort, intersections or unions of set terms of the same sort, or variables).

We also write $\{\}$ as $\emptyset$. Note that symbols like $\cup$ are overloaded to denote different functions for each sort. To avoid confusion, we sometimes use subscripts to emphasise the sort, as in $\emptyset_{\{\tau\}}$. A term is *basic* if it does not contain $\cap$ or $\cup$, and *ground* if it is variable-free.

Every predicate $p \in \mathbf{P}$ is associated with a sort $\mathsf{sort}(p)$. A *schema* $\mathbf{S} = \langle \mathbf{P}^{\mathrm{EDB}}, \mathbf{P}^{\mathrm{IDB}} \rangle$ is a partition of $\mathbf{P}$ into *EDB predicates* $\mathbf{P}^{\mathrm{EDB}}$and *IDB predicates* $\mathbf{P}^{\mathrm{IDB}}$. An *atom* for predicate $p \in \mathbf{P}$ is an expression $p(t)$ where $t \in \mathbf{T}_{\mathsf{sort}(p)}$. If p is of a tuple sort, we write $p(t_1, \ldots, t_\ell)$ instead of $p(\langle t_1, \ldots, t_\ell \rangle)$. Similarly, we may omit the outermost parentheses in set terms $(t_1 \cup t_2)$ and $(t_1 \cap t_2)$, writing $t_1 \cup t_2$ and $t_1 \cap t_2$, respectively. A $\mathsf{Datalog}^{\mathrm{CV}}$ *fact* is an atom $p(t)$, where t contains only basic ground terms. A $\mathsf{Datalog}^{\mathrm{CV}}$ *rule* is a formula of the form $\forall \boldsymbol{x}.\varphi \to \psi$ where the *body* φ and the *head* ψ are conjunctions of atoms, and $\boldsymbol{x}$ is a list of all variables in the rule. We allow φ to be empty, but require ψ to contain at least one atom. Universal quantifiers are usually omitted. A $\mathsf{Datalog}^{\mathrm{CV}}$ *program* $\mathbb{P}$ for schema $\mathbf{S}$ is a finite set of $\mathsf{Datalog}^{\mathrm{CV}}$ rules, where EDB predicates do not occur in rule heads. A $\mathsf{Datalog}^{\mathrm{CV}}$ *database* $\mathbb{D}$ for schema $\mathbf{S}$ is a finite set of facts using only EDB predicates. A $\mathsf{Datalog}^{\mathrm{CV}}$ *BCQ* is a sentence of the form $\exists \boldsymbol{x}.\varphi[\boldsymbol{x}]$, where φ is a conjunction of atoms containing only basic terms.

Already before defining the semantics of $\mathsf{Datalog}^{\mathrm{CV}}$ formally, we can foster our intuitive understanding of these definitions by considering some examples. This will also yield some useful insights into the expressive power of this formalism.

▶ **Example 2.** An operation that is commonly studied in query formalisms with complex values is the powerset function. We can capture this in atoms of the form $\mathsf{PS}_\sigma(S, P)$ expressing that a set S (of set sort $\sigma = \{\tau\}$) has the powerset P (of sort $\{\sigma\} = \{\{\tau\}\}$). We use auxiliary predicates PSU_τ of sort $\langle \tau, \{\{\tau\}\}, \{\{\tau\}\} \rangle$ where an atom $\mathsf{PSU}_\tau(t, P, Q)$ expresses, informally speaking, that $Q = \{S \cup \{t\} \mid S \in P\}$. As usual, we omit universal quantifiers.

$$\to \mathsf{PSU}_\tau(x, \emptyset, \emptyset) \tag{4}$$

$$\mathsf{PSU}_\tau(x, P, Q) \to \mathsf{PSU}_\tau(x, P \cup \{S\}, Q \cup \{S \cup \{x\}\}) \tag{5}$$

$$\to \mathsf{PS}_\sigma(\emptyset, \{\emptyset\}) \tag{6}$$

$$\mathsf{PS}_\sigma(S, P) \wedge \mathsf{PSU}_\tau(x, P, Q) \to \mathsf{PS}_\sigma(S \cup \{x\}, P \cup Q) \tag{7}$$

Rule (6) states that the powerset of $\emptyset$ is $\{\emptyset\}$ and rule (7) states that, given the powerset P of S, the powerset of $S \cup \{x\}$ is obtained from P by adding $Q \cup \{x\}$ for every $Q \in P$.

On a database containing only the constant c, the following facts are entailed:

$$\mathsf{PSU}_\tau(c,\emptyset,\emptyset) \qquad \mathsf{PSU}_\tau(c,\{\emptyset\},\{\{c\}\}) \qquad \mathsf{PSU}_\tau(c,\{\{\emptyset\}\},\{\emptyset_c\})$$
$$\mathsf{PS}_\tau(\emptyset,\{\emptyset\}) \qquad \mathsf{PS}_\tau(\{c\},\{\emptyset,\{c\}\})$$

Note that rules (4) and (5) are "unsafe": they use variables in the head that do not occur in the body. Our semantics restricts the scope of variables to a finite active domain so that this is never a problem. Using auxiliary predicates AD_τ and additional rules, we can transform any such rule into a safe rule: for every predicate $p \in \mathbf{P}^{\mathrm{EDB}}$ of sort τ, we add a rule $p(x) \to \mathsf{AD}_\tau(x)$. Furthermore, for tuple sorts $\tau = \langle\tau_1,\dots,\tau_\ell\rangle$, we add rules $\mathsf{AD}_\tau(x_1,\dots,x_\ell) \to \mathsf{AD}_{\tau_k}(x_k)$ for $1 \le k \le \ell$ and $\mathsf{AD}_{\tau_1}(x_1) \wedge \cdots \mathsf{AD}_{\tau_\ell}(x_\ell) \to \mathsf{AD}_\tau(x_1,\dots,x_\ell)$; for set sorts $\tau = \{\sigma\}$, we add rules $\mathsf{AD}_\sigma(x) \to \mathsf{AD}_\tau(\{x\})$, $\mathsf{AD}_\tau(\{x\}) \to \mathsf{AD}_\sigma(x)$, and $\mathsf{AD}_\tau(x) \wedge \mathsf{AD}_\tau(y) \to \mathsf{AD}_\tau(x \cap y) \wedge \mathsf{AD}_\tau(x \cup y)$. An unsafe variable v of sort τ in rule ρ can then be eliminated by adding an atom $\mathsf{AD}_\tau(v)$ to the body of ρ. For (4) and (5), we would thus obtain the following rules:

$$\mathsf{AD}_\tau(x) \to \mathsf{PSU}_\tau(x,\emptyset,\emptyset) \tag{8}$$
$$\mathsf{AD}_\tau(x) \wedge \mathsf{AD}_{\{\tau\}}(S) \wedge \mathsf{PSU}_\tau(x,P,Q) \to \mathsf{PSU}_\tau(x,P \cup \{S\},Q \cup \{S \cup \{x\}\}) \tag{9}$$

▶ **Example 3.** We can also define further set-related predicates and functions. The following rules show how to define predicates $\subseteq_\sigma$, $\in_\sigma$, $\notin_\sigma$, $\neq_\tau$, and $\subset_\sigma$ for a set sort σ and arbitrary sort τ, where we write infix $t_1 \diamond t_2$ instead of $\diamond(t_1,t_2)$ for better readability:

$$\to S \subseteq_\sigma S \cup T \tag{10}$$
$$S \cup \{x\} \subseteq_\sigma S \to x \in_\sigma S \tag{11}$$
$$S \cap \{x\} \subseteq_\sigma \emptyset \to x \notin_\sigma S \tag{12}$$
$$\{x\} \cap \{y\} \subseteq_{\{\tau\}} \emptyset \to x \neq_\tau y \tag{13}$$
$$S \subseteq_\sigma T \wedge x \in_\sigma T \wedge x \notin_\sigma S \to S \subset_\sigma T \tag{14}$$

We may therefore assume without loss of generality that the shortcuts defined in Examples 2 and 3 are always available in $\mathsf{Datalog}^{\mathrm{CV}}$ programs. Similarly, unless stated to the contrary, we assume throughout the paper that databases contain only facts of sort Δ or of some tuple sort $\langle\Delta,\dots,\Delta\rangle$, since all other facts can be constructed using appropriate rules.

Semantics

As usual for Datalog, we consider a *Herbrand interpretation* whose domain is the set of constants that syntactically occur in a given program and database, where we interpret constants as themselves (*unique name assumption*). To this end, we interpret sorts and ground terms with a function $\mathsf{eval}(\cdot)$. For sorts, let $\mathsf{eval}(\Delta) := \mathbf{C}$, $\mathsf{eval}(\langle\tau_1,\dots,\tau_\ell\rangle) := \mathsf{eval}(\tau_1) \times \cdots \times \mathsf{eval}(\tau_\ell)$, and $\mathsf{eval}(\{\tau\}) := \mathcal{P}(\mathsf{eval}(\tau))$ where $\mathcal{P}(S)$ denotes the set of all subsets of S. For ground terms t, we recursively define $\mathsf{eval}(t)$ as follows:

- $\mathsf{eval}(c) := c$ for $c \in \mathbf{C}$;
- $\mathsf{eval}(\{t_1,\dots,t_n\}) := \{\mathsf{eval}(t_1),\dots,\mathsf{eval}(t_n)\}$;
- $\mathsf{eval}(\langle s_1,\dots,s_\ell\rangle) := \langle\mathsf{eval}(s_1),\dots,\mathsf{eval}(s_\ell)\rangle$;
- $\mathsf{eval}(t_1 \cap t_2) := \mathsf{eval}(t_1) \cap \mathsf{eval}(t_2)$; and
- $\mathsf{eval}(t_1 \cup t_2) := \mathsf{eval}(t_1) \cup \mathsf{eval}(t_2)$.

The *domain* $\mathsf{dom}(\mathbb{P},\mathbb{D})$ of a $\mathsf{Datalog}^{\mathrm{CV}}$ program $\mathbb{P}$ and database $\mathbb{D}$ is the set of all constants that occur in $\mathbb{P}$ or $\mathbb{D}$. An *interpretation* $\mathcal{I}$ for $\mathbb{P}$ and $\mathbb{D}$ maps every predicate p to a set

$p^{\mathcal{I}} \subseteq \mathsf{eval}(\mathsf{sort}(p))$ that contains only constants from $\mathsf{dom}(\mathbb{P}, \mathbb{D})$. $\mathcal{I}$ *satisfies* a ground atom $p(t)$, written $\mathcal{I} \models p(t)$, if $\mathsf{eval}(t) \in p^{\mathcal{I}}$; it *satisfies* a conjunction φ of such atoms, written $\mathcal{I} \models \varphi$, if $\mathcal{I} \models \alpha$ for all α in φ; and it *satisfies* a variable-free rule $\varphi \to \psi$, written $\mathcal{I} \models \varphi \to \psi$, if $\mathcal{I} \not\models \varphi$ or $\mathcal{I} \models \psi$. $\mathcal{I}$ *satisfies* $\mathbb{P}$ if $\mathcal{I} \models \rho'$ holds for every *ground instance* ρ' of ρ, i.e., for every variable-free rule ρ' obtained from a rule $\rho \in \mathbb{P}$ by replacing each variable $v \in \mathbf{V}_\tau$ in ρ with a ground term $t_v \in \mathsf{eval}(\tau)$ that uses only constants from $\mathsf{dom}(\mathbb{P}, \mathbb{D})$. $\mathcal{I}$ *satisfies* $\mathbb{D}$ if $\mathcal{I} \models \alpha$ for all $\alpha \in \mathbb{D}$. If $\mathcal{I}$ satisfies X, we also say that $\mathcal{I}$ is a *model* of X.

This model theory gives rise to entailment as usual: a ground fact α is entailed by $\mathbb{P}$ and $\mathbb{D}$, written $\mathbb{P}, \mathbb{D} \models \alpha$, if $\mathcal{I} \models \alpha$ for all models $\mathcal{I}$ of $\mathbb{P}$ and $\mathbb{D}$. BCQ entailment can be reduced to ground fact entailment by using BCQs as bodies of a rule with some ground head atom that is not derived by any other rule. As in the case of Datalog, entailment can be decided by considering a single *least model*, which could also be computed by a chase-like process, as the following theorem shows.

▶ **Theorem 4.** *Let* $\mathbf{S}$ *be a schema,* $\mathbb{P}$ *be a* $\mathsf{Datalog}^{CV}$ *program for* $\mathbf{S}$*, and* $\mathbb{D}$ *be a* $\mathsf{Datalog}^{CV}$ *database for* $\mathbf{S}$*. Then* $\mathbb{P}, \mathbb{D}$ *has a unique* least model $\mathcal{I}$ *such that for all ground facts,* $\mathcal{I} \models \alpha$ *iff* $\mathbb{P}, \mathbb{D} \models \alpha$*. Furthermore,* $\mathcal{I}$ *is the least fixed point of the* immediate consequence operator $T_{\mathbb{P}}$ *applied to* $\mathbb{D}$*: for a set* S *of ground facts,* $T_{\mathbb{P}}(S) \coloneqq S \cup \{\mathrm{head}(\rho') \mid \mathrm{body}(\rho') \in S$ *for a ground instance* ρ' *of* $\rho \in \mathbb{P}\}$ *is the union of* S *and the set of ground facts obtained by adding all heads of ground instances of rules that have their bodies contained in* S*.*

Complexity

The complexity of deciding entailment for a $\mathsf{Datalog}^{\mathrm{CV}}$ program $\mathbb{P}$ depends crucially on the set sorts occurring in $\mathbb{P}$, motivating the following definition.

▶ **Definition 5.** *The* set height $\mathsf{s\text{-}height}(\tau)$ *of a sort* τ *is defined recursively:*

- $\mathsf{s\text{-}height}(\Delta) \coloneqq 0$*;*
- $\mathsf{s\text{-}height}(\{\!|\sigma|\!\}) \coloneqq \mathsf{s\text{-}height}(\sigma) + 1$*; and*
- $\mathsf{s\text{-}height}(\langle \sigma_1, \ldots, \sigma_\ell \rangle) \coloneqq \max_{1 \le i \le \ell} \mathsf{s\text{-}height}(\sigma_i)$.

Analogously, the tuple height $\mathsf{t\text{-}height}(\tau)$ *of a sort* τ *is*

- $\mathsf{t\text{-}height}(\Delta) \coloneqq 0$*;*
- $\mathsf{t\text{-}height}(\{\!|\sigma|\!\}) \coloneqq \mathsf{t\text{-}height}(\sigma)$*; and*
- $\mathsf{t\text{-}height}(\langle \tau_1, \ldots, \tau_\ell \rangle) \coloneqq 1 + \max_{1 \le i \le \ell} \mathsf{t\text{-}height}(\tau_i)$.

The set height (tuple height) of a term, predicate, or variable is the set height (tuple height) of its sort, the set height (tuple height) of a schema $\mathbf{S}$ *is the largest set height (tuple height) of any predicate in* $\mathbf{S}$*, and the* height *of* $\mathbf{S}$ *is the sum of its set height and its tuple height.*

We get the following lower bounds for reasoning in $\mathsf{Datalog}^{\mathrm{CV}}$, where 0ExpTime denotes PTime. Matching upper bounds are shown in Section 4.

▶ **Theorem 6.** *Consider a* $\mathsf{Datalog}^{CV}$ *program* $\mathbb{P}$ *and a database* $\mathbb{D}$ *for schema* $\mathbf{S}$*, and a ground fact* α*. Deciding* $\mathbb{P}, \mathbb{D} \models \alpha$ *is* kExpTime*-hard with respect to the size of* $\mathbb{D}$ *(*data complexity*) and* $(k+2)$ExpTime*-hard with respect to the size of* $\mathbb{D}$ *and* $\mathbb{P}$ *(*combined complexity*), where* $k = \mathsf{s\text{-}height}(\mathbf{S})$.

If $\mathsf{t\text{-}height}(\mathbf{S})$ *is bounded, combined complexity is only* $(k+1)$ExpTime*-hard.*

Since Datalog has bounded tuple height 1 and set height 0, our complexity bounds (PTime data and ExpTime combined) are optimal in this case [16].

Proof. For data complexity, we reduce from the halting problem for k-exponentially time-bounded (in the size of $\mathbb{D}$) Turing machines on the empty tape, where $\mathbb{P}$ encodes the Turing machine. The main challenge is the construction of the tape, as Turing machines can be simulated already in Datalog [16].

Assuming that a linear order $s_1 < s_2 < \cdots < s_\ell$ for sort τ is encoded using unary predicates $\mathsf{first}_\tau(s_1)$ and $\mathsf{last}_\tau(s_\ell)$ and a binary predicate $\mathsf{next}_\tau(s_i, s_{i+1})$, the rules in Figure 1 derive an exponentially longer chain for sort $\sigma = \{\tau\}$. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_\ell\}$.

Intuitively, the program enumerates the power set $\mathcal{P}(\mathcal{S})$ of $\mathcal{S}$ in lexicographic order, producing the exponentially longer linear order $\emptyset < \{s_1\} < \{s_1, s_2\} < \cdots < \{s_1, s_2, \ldots, s_\ell\} < \{s_2\} < \cdots < \{s_2, \ldots, s_\ell\} < \cdots < \{s_\ell\}$. A fact $\mathsf{mkStep}_\sigma(S, t, x)$ will lead to next_σ-facts corresponding to the segment $S \cup \{x\} < \cdots < S \cup \{s_\ell\}$, with t tracking the least element not contained in S. Similarly, a fact $\mathsf{step}_\sigma(S, t, x, T)$ corresponds to the segment $S < S \cup \{x\} < \cdots < T$, where t is again the least element not contained in S.

To see that rules (15)–(19) are correct, we show the following claim: for any set $S \subseteq \mathcal{S}$ and any $t, x \in \{s \in \mathcal{S} \mid \forall y \in S.\ y < t \wedge y < x\}$, the fact $\mathsf{mkStep}_\sigma(S, t, x)$ will result in facts $\mathsf{next}_\sigma(S \cup \{x\}, \ldots), \ldots, \mathsf{next}_\sigma(\ldots, S \cup \{s_\ell\})$ corresponding to the segment $S \cup \{x\} < \cdots < S \cup \{s_\ell\}$ of the lexicographic order on $\mathcal{P}(\mathcal{S})$, and furthermore a fact $\mathsf{step}_\sigma(S, t, x, S \cup \{s_\ell\})$.

We show the claim by induction on the third position in mkStep_σ-facts. For the base case, consider $\mathsf{mkStep}_\sigma(S, s_\ell, s_\ell)$. From rule (16), we immediately get $\mathsf{next}_\sigma(S, S \cup \{s_\ell\})$, and rule (17) derives $\mathsf{step}_\sigma(S, s_\ell, s_\ell, S \cup \{s_\ell\})$. For the induction step, suppose that x has the direct successor y. Rule (18) derives the facts $\mathsf{mkStep}_\sigma(S \cup \cup\, x, y, y)$ and $\mathsf{mkStep}_\sigma(S, t, y)$. By the induction hypothesis, we therefore derive next_σ and step_σ-facts corresponding to the segments $S \cup \{x\} < S \cup \{x, y\} < \cdots < S \cup \{x, s_\ell\}$ of successors containing x, and $S \cup \{y\} < \cdots < S \cup \{s_\ell\}$ of successors not containing x. Since y is the direct successor of x, the immediate successor of $S \cup \{x, s_\ell\}$ is $S \cup \{y\}$, and rule (19) therefore connects the two segments. Lastly, whenever $t = x$, rule (16) makes $S \cup \{x\}$ the direct successor of S. Such mkStep_σ-facts are derived by rule (15), where we immediately see that this is correct, since $\{s_1\}$ is indeed the direct successor of $\emptyset_\sigma$, and by rule (18), where we consider $S \cup \{x\}$. Since y is the direct successor of x, this makes $S \cup \{x, y\}$ the direct successor of $S \cup \{x\}$.

Since no other mkStep_σ-facts are derived, no cycles in next_σ-facts can appear, and thus we indeed obtain a linear order. Correctness then follows since rule (15) derives $\mathsf{first}_\sigma(\emptyset_\sigma)$, $\mathsf{mkStep}_\sigma(\emptyset_\sigma, s_1, s_1)$, and $\mathsf{last}_\sigma(\{s_\ell\})$.

By instantiating these rules for multiple set sorts, we can thus construct k-exponentially long linear orders from a linear order in the input database, which we can use as a Turing tape for our simulation [16].

For combined complexity, we use a tuple sort of nested tuples of height t and constants at the lowest level. We then construct a chain of doubly-exponential length in t by ordering tuples at each of the t levels lexicographically, starting from the length-ℓ chain of database constants. If tuples have arity $\geq a$, this yields a chain of length $\geq \ell^{a^t}$. The construction then proceeds as before, using this chain as a basis for an even longer chain of sets. Note that if t is bounded by a constant (but a is not), then ℓ^{a^t} is merely single exponential. ◀

4 From Complex Values to TGDs

In this section, we reduce query answering over $\mathsf{Datalog}^{\mathrm{CV}}$ programs and $\mathsf{Datalog}^{\mathrm{CV}}$ databases to query answering over sets of TGDs and (unsorted) databases for which the standard chase terminates reliably (i.e., under all strategies).

$$\mathsf{first}_\tau(x) \wedge \mathsf{last}_\tau(z) \to \mathsf{first}_\sigma(\emptyset_\sigma) \wedge \mathsf{mkStep}_\sigma(\emptyset_\sigma, x, x) \wedge \mathsf{last}_\sigma(\{z\}) \quad (15)$$

$$\mathsf{mkStep}_\sigma(S, t, t) \to \mathsf{next}_\sigma(S, S \cup \{t\}) \quad (16)$$

$$\mathsf{mkStep}_\sigma(S, t, x) \wedge \mathsf{last}_\tau(x) \to \mathsf{step}_\sigma(S, t, x, S \cup \{x\}) \quad (17)$$

$$\mathsf{mkStep}_\sigma(S, t, x) \wedge \mathsf{next}_\tau(x, y) \to \mathsf{mkStep}_\sigma(S \cup \{x\}, y, y) \wedge \mathsf{mkStep}_\sigma(S, t, y) \quad (18)$$

$$\mathsf{mkStep}_\sigma(S, t, x) \wedge \mathsf{next}_\tau(x, y) \wedge \mathsf{step}_\sigma(S \cup \{x\}, y, y, X) \wedge \mathsf{step}_\sigma(S, t, y, Z) \to \mathsf{next}_\sigma(X, S \cup \{y\}) \wedge \mathsf{step}_\sigma(S, t, x, Z) \quad (19)$$

Figure 1 A $\mathsf{Datalog}^{\mathrm{CV}}$ program for a set sort $\sigma = \{\tau\}$ deriving an exponentially long linear order

$$\textstyle\bigwedge_{i=1}^{\ell} \mathsf{sort}_{\tau_i}(x_i) \to \exists z.\mathsf{tuple}_\pi(z, x_1, \ldots, x_\ell) \wedge \mathsf{sort}_\pi(z) \quad (20)$$

$$\to \exists V.\mathsf{empty}_\sigma(V) \wedge \mathsf{sort}_\sigma(V) \wedge \mathsf{done}_\sigma(V) \quad (21)$$

$$\mathsf{done}_\sigma(V) \wedge \mathsf{sort}_\tau(x) \to \exists W.\mathsf{SU}_\sigma(x, V, W) \wedge \mathsf{sort}_\sigma(W) \wedge \mathsf{todo}_\sigma(W, W) \quad (22)$$

$$\mathsf{todo}_\sigma(V, W) \wedge \mathsf{SU}_\sigma(x, U, V) \to \mathsf{SU}_\sigma(x, W, W) \wedge \mathsf{todo}_\sigma(U, W) \quad (23)$$

$$\mathsf{todo}_\sigma(V, W) \wedge \mathsf{empty}_\sigma(V) \to \mathsf{done}_\sigma(W) \quad (24)$$

Figure 2 TGDs for axiomatising $\mathsf{Datalog}^{\mathrm{CV}}$ sorts and terms; we instantiate (20) for sorts $\pi = \langle \tau_1, \ldots, \tau_\ell \rangle$, and (21)–(24) for sorts $\sigma = \{\tau\}$

We build upon prior work by Carral et al. [13], which established a translation from "Datalog with Sets" Datalog(S) to sets of TGDs. Datalog(S) can be seen as the restriction of $\mathsf{Datalog}^{\mathrm{CV}}$ to schemas where every predicate has sort Δ, $\{\Delta\}$, or $\langle \tau_1, \ldots, \tau_\ell \rangle$ with $\tau_i \in \{\Delta, \{\Delta\}\}$ for $1 \leq \ell$, i.e., where every position is either an element of the domain or a set over such elements. The key idea is to represent sets by recursively constructing them as unions of singletons and smaller sets. While our translation shares this approach, an important difference is that our translation produces sets of TGDs for which the standard chase terminates on any database, irregardless of the order of rule applications, while the translation of Carral et al. relies on the prioritisation of certain rules to ensure termination.

The translation consists of two parts: a set of auxiliary rules that axiomatise the semantics of set functions and predicates for all sorts used in the program, and, for every given rule, a rewritten rule that uses these defined predicates instead of set functions. After the translation, all $\mathsf{Datalog}^{\mathrm{CV}}$ terms are represented by individual nulls or constants. For example, facts $\mathsf{tuple}_\pi(r, t_1, \ldots, t_\ell)$ express that r represents the tuple $\langle t_1, \ldots, t_\ell \rangle$ of sort π. To encode sets, we use facts $\mathsf{SU}_\sigma(x, S, T)$ ("singleton union"), which can be read as "$\{x\} \cup_\sigma S = T$." Together with a representative $c_{\emptyset_\sigma}$ for $\emptyset_\sigma$, this allows us to represent every set $\{e_1, \ldots, e_n\}$ by a term c_n for which we give a list of facts $\mathsf{SU}_\sigma(e_1, c_{\emptyset_\sigma}, c_1), \ldots, \mathsf{SU}_\sigma(e_n, c_{n-1}, c_n)$. Note that this representation is not unique; in general we have indeed multiple representatives for every set. This may, in turn, lead to multiple representatives for the same tuple if sets occur somewhere in the component sorts. To ensure that this does not pose problems, our translated programs will include congruence rules that propagate derived facts between different representatives of the same value.

The rules for axiomatising these basic predicates are as given in Figure 2, which also defines $\mathsf{empty}_\sigma(t)$ ("t represents $\emptyset_\sigma$") and $\mathsf{sort}_\tau(t)$ ("t represents a term of sort τ"). We assume (and will ensure) that facts for sort_Δ are defined for the constants in the active domain. Rule (20) creates representatives for π-tuples. The remaining rules (21)–(24) create

$$\mathsf{sort}_\sigma(V) \wedge \mathsf{sort}_\sigma(W) \to \mathsf{ckSub}_\sigma(V,V,W) \tag{25}$$
$$\mathsf{ckSub}_\sigma(U,V,W) \wedge \mathsf{SU}_\sigma(x,U',U) \wedge \mathsf{SU}_\sigma(x,V,V) \to \mathsf{ckSub}_\sigma(U',V,W) \tag{26}$$
$$\mathsf{ckSub}_\sigma(U,V,W) \wedge \mathsf{empty}_\sigma(U) \to \mathsf{subset}_\sigma(V,W) \tag{27}$$
$$\mathsf{subset}_\sigma(V,W) \wedge \mathsf{subset}_\sigma(W,V) \to \mathsf{eq}_\sigma(V,W) \tag{28}$$
$$\mathsf{empty}_\sigma(V) \wedge \mathsf{sort}_\sigma(W) \to \mathsf{U}_\sigma(V,W,W) \tag{29}$$
$$\mathsf{SU}_\sigma(x,W,W') \wedge \mathsf{U}_\sigma(V,W',U) \wedge \mathsf{SU}_\sigma(x,V,V') \to \mathsf{U}_\sigma(V',W,U) \tag{30}$$
$$\mathsf{sort}_\sigma(V) \wedge \mathsf{sort}_\tau(x) \to \mathsf{ckNIn}_\sigma(V,x,V) \tag{31}$$
$$\mathsf{ckNIn}_\sigma(U,x,V) \wedge \mathsf{SU}_\sigma(y,U',U) \wedge \mathsf{NEq}_\tau(x,y) \to \mathsf{ckNIn}_\sigma(U',x,V) \tag{32}$$
$$\mathsf{ckNIn}_\sigma(U,x,V) \wedge \mathsf{empty}_\sigma(U) \to \mathsf{NIn}_\sigma(x,V) \tag{33}$$
$$\mathsf{tuple}_\pi(z,x_1,\dots,x_\ell) \wedge \mathsf{tuple}_\pi(z',y_1,\dots,y_\ell) \wedge \mathsf{NEq}_{\tau_i}(x_i,y_i) \to \mathsf{NEq}_\pi(z,z') \tag{34}$$
$$\mathsf{SU}_\sigma(x,V,V) \wedge \mathsf{NIn}_\sigma(x,W) \to \mathsf{NEq}_\sigma(V,W) \wedge \mathsf{NEq}_\sigma(W,V) \tag{35}$$
$$\mathsf{empty}_\sigma(V) \wedge \mathsf{sort}_\sigma(W) \to \mathsf{I}_\sigma(V,W,V) \tag{36}$$
$$\mathsf{I}_\sigma(U,V,W) \wedge \mathsf{SU}_\sigma(x,U,U') \wedge \mathsf{NIn}_\sigma(x,V) \to \mathsf{I}_\sigma(U',V,W) \tag{37}$$
$$\mathsf{I}_\sigma(U,V,W) \wedge \mathsf{SU}_\sigma(x,U,U') \wedge \mathsf{SU}_\sigma(x,V,V') \wedge \mathsf{SU}_\sigma(x,W,W') \to \mathsf{I}_\sigma(U',V',W') \tag{38}$$

Figure 3 TGDs for axiomatising $\mathsf{Datalog}^{\mathrm{CV}}$ sort functions and predicates; we instantiate (34) for sorts $\pi = \langle \tau_1, \dots, \tau_\ell \rangle$ and all $1 \leq i \leq \ell$, and all other rules for sorts $\sigma = \{\tau\}$

representatives for sets by generalising an approach that we outlined in our previous work [26] to arbitrary set sorts $\sigma = \{\tau\}$: Rule (21) creates a unique representative for $\emptyset_\sigma$, which is immediately marked as "done". Given any "done" set V and an element x of sort τ, rule (22) introduces a representative W for $S \cup \{x\}$, which is marked as "todo". Then rule (23) derives facts representing $W = U \cup \{x\}$ for all sets $U \subseteq V$ and all $x \in U$. Only when all such facts have been derived is W marked "done" (rule (24)). But since rule (22) is only applicable for "done" sets, it will always be blocked for sets V and elements x with $x \in V$. Thus, it can only be applied finitely often, and termination is therefore guaranteed.

The essential correctness claim for these rules is as follows:

▶ **Lemma 7.** *Let Σ be the set of rules (21)–(24) for a sort $\sigma = \{\tau\}$. For every database $\mathcal{D}$ that contains only facts of the form $\mathsf{sort}_\tau(c)$, every standard chase sequence over Σ and $\mathcal{D}$ terminates after $O(2^{|\mathcal{D}|})$ many steps, producing a finite result $\mathcal{I}$ that is unique up to isomorphism. For this result $\mathcal{I}$ and the sets $S_t := \{c \mid \mathsf{SU}_\sigma(c,t,t) \in \mathcal{I}\}$, we have:*

- *if $\mathsf{empty}_\sigma(t) \in \mathcal{I}$ then $S_t = \emptyset$,*
- *if $\mathsf{SU}_\sigma(c,t,u) \in \mathcal{I}$ then $S_u = S_t \cup \{c\}$, and*
- *$\{S_t \mid \mathsf{sort}_\sigma(t) \in \mathcal{I}\}$ is the powerset of $\{c \mid \mathsf{sort}_\tau(c) \in \mathcal{D}\}$.*

To complete the translation, we use further facts $\mathsf{eq}_\sigma(t_1,t_2)$ ("$t_1 = t_2$"), $\mathsf{U}_\sigma(t_1,t_2,t)$ ("$t_1 \cup t_2 = t$"), and $\mathsf{I}_\sigma(t_1,t_2,t)$ ("$t_1 \cap t_2 = t$"), axiomatised in Figure 3. Expressing eq is necessary since our modelling in Figure 2 may lead to several representatives for the same set. To reconcile this, we compute facts for subset_σ: for every pair $\langle V, W \rangle$ of sets (25), we iterate over elements of V (26) to verify that they are in W, until all elements have been processed (27). Equality follows from mutual inclusion (28).

Unions are defined from singleton unions by a simple recursion (29)–(30). To compute intersections, we define inequality NEq_τ (for all sorts τ) and non-containment NIn_σ (for

set sorts σ) by mutual recursion (31)–(35). Facts for NEq_Δ will be defined explicitly for constants of the active domain by our translation. Rules (31)–(33) are similar to rules (25)–(27). Deriving inequalities for tuples and sets is then straightforward (34)–(35). Finally, intersections are defined starting from the empty set (36), and recursively adding elements that are missing/present in the other set (37)/(38). Reusing the notation S_t from Lemma 7, we state the essential correctness results for this translation:

▶ **Lemma 8.** *Let Σ be the set of rules (21)–(24) and (25)–(30) for a sort $\sigma = \{\!\{\tau\}\!\}$. For every database $\mathcal{D}$ that contains only facts of the form $\mathsf{sort}_\tau(c)$, the standard chase over Σ and $\mathcal{D}$ produces a finite result $\mathcal{I}$ that is unique up to isomorphism, such that:*

- $\mathsf{eq}_\sigma(t,u) \in \mathcal{I}$ *iff* $S_t = S_u$,
- $\mathsf{U}_\sigma(t,u,v) \in \mathcal{I}$ *implies* $S_t \cup S_u = S_v$, *and*
- *for all* $\mathsf{sort}_\sigma(t), \mathsf{sort}_\sigma(u) \in \mathcal{I}$, *there is some fact* $\mathsf{U}_\sigma(t,u,v) \in \mathcal{I}$.

▶ **Lemma 9.** *Let Σ be the set of rules (21)–(24), (31)–(33), and (35)–(38) for a sort $\sigma = \{\!\{\tau\}\!\}$. For every database $\mathcal{D}$ of the form $\{\mathsf{sort}_\tau(c) \mid c \in \Pi\} \cup \{\mathsf{NEq}_\tau(c,d) \mid c \neq d; c,d \in \Pi\}$ for some finite set $\Pi \subseteq \mathbf{C}$ of constants, the standard chase over Σ and $\mathcal{D}$ produces a finite result $\mathcal{I}$ that is unique up to isomorphism, such that:*

- $\mathsf{NIn}_\sigma(c,t) \in \mathcal{I}$ *iff* $c \notin S_t$,
- $\mathsf{NEq}_\sigma(t,u) \in \mathcal{I}$ *iff* $S_t \neq S_u$,
- $\mathsf{I}_\sigma(t,u,v) \in \mathcal{I}$ *implies* $S_t \cap S_u = S_v$, *and*
- *for all* $\mathsf{sort}_\sigma(t), \mathsf{sort}_\sigma(u) \in \mathcal{I}$, *there is some fact* $\mathsf{I}_\sigma(t,u,v) \in \mathcal{I}$.

To translate rules in $\mathbb{P}$, we associate with every non-constant term $t \in \mathbf{T}_\tau$ a distinct variable $x_t \in \mathbf{V}_\tau$. Let $\mathsf{tr}(t) \coloneqq t$ if $t \in \mathbf{C}$ and set $\mathsf{tr}(t) \coloneqq x_t$ otherwise, and define a set of atoms $\mathsf{flat}(t)$ for terms t of sort τ recursively as follows:

- if $t \in \mathbf{C} \cup \mathbf{V}_\tau$ then $\mathsf{flat}(t) \coloneqq \{\mathsf{sort}_\tau(\mathsf{tr}(t))\}$;
- if $t = \langle t_1, \dots, t_\ell \rangle$ then $\mathsf{flat}(t) \coloneqq \{\mathsf{tuple}_\tau(x_t, \mathsf{tr}(x_{t_1}), \dots, \mathsf{tr}(x_{t_\ell}))\} \cup \bigcup_{i=1}^{\ell} \mathsf{flat}(t_i)$;
- if $t = \emptyset_\tau$ then $\mathsf{flat}(t) \coloneqq \{\mathsf{empty}(x_t)\}$;
- if $t = \{\!\{t_1, \dots, t_n\}\!\}$ then $\mathsf{flat}(t) \coloneqq \{\mathsf{SU}(\mathsf{tr}(t_1), x_s, x_t)\} \cup \mathsf{flat}(t_1) \cup \mathsf{flat}(s)$ for $s = \{\!\{t_2, \dots, t_n\}\!\}$ (which can be empty);
- if $t = t_1 \cup t_2$ then $\mathsf{flat}(t) \coloneqq \{\mathsf{U}_\tau(\mathsf{tr}(t_1), \mathsf{tr}(t_2), x_t)\} \cup \mathsf{flat}(t_1) \cup \mathsf{flat}(t_2)$; and
- if $t = t_1 \cap t_2$ then $\mathsf{flat}(t) \coloneqq \{\mathsf{I}_\tau(\mathsf{tr}(t_1), \mathsf{tr}(t_2), x_t)\} \cup \mathsf{flat}(t_1) \cup \mathsf{flat}(t_2)$.

To translate a rule ρ to a TGD $\mathsf{tr}(\rho)$, replace each term t in ρ by $\mathsf{tr}(t)$ and add $\mathsf{flat}(t)$ to the body of ρ. For a fact $p(t)$, let $\mathsf{tr}(p(t)) \coloneqq \{p(\mathsf{tr}(t))\} \cup \mathsf{flat}(t)$, and for a conjunction $\varphi = \bigwedge_{i=1}^{n} \varphi_i$, let $\mathsf{tr}(\varphi) \coloneqq \bigcup_{i=1}^{n} \mathsf{tr}(\varphi_i)$. A BCQ $\exists \boldsymbol{x}.\varphi[\boldsymbol{x}]$ is translated into the BCQ obtained by existentially quantifying all variables in $\mathsf{tr}(\varphi)$.

For $\mathsf{Datalog}^{\mathrm{CV}}$ program $\mathbb{P}$, let $\mathsf{tr}(\mathbb{P})$ consist of (1) $\mathsf{tr}(\rho)$ for all $\rho \in \mathbb{P}$, (2) all rules in Figures 2 and 3, instantiated for all subsorts of sorts in $\mathbb{P}$, (3) the *congruence rules* $\to \mathsf{eq}(x,x)$ and $p(\boldsymbol{x}) \wedge \mathsf{eq}(x_i, x_i') \to p(\boldsymbol{x}|_{i \mapsto x_i'})$ for all predicates p and all $1 \leq i \leq \mathsf{ar}(p)$, and (4) the facts $\{\mathsf{sort}_\Delta(c) \mid c \in \Pi\} \cup \{\mathsf{NEq}_\Delta(c,d) \mid c \neq d; c,d \in \Pi\}$ for Π the set of constants in $\mathbb{P}$. Likewise, let $\mathsf{tr}(\mathbb{D})$ consist of all facts obtained by uniformly replacing, for each $\mathsf{Datalog}^{\mathrm{CV}}$ fact $\alpha \in \mathbb{D}$, the variables in $\mathsf{tr}(\alpha)$ with fresh constants. We note that tr can be computed in polynomial time (even in logarithmic space) in all cases. We obtain the following correctness result:

▶ **Theorem 10.** *For every $\mathsf{Datalog}^{CV}$ program $\mathbb{P}$ and database $\mathbb{D}$ for $\mathbb{P}$:*

1. *for every $\mathsf{Datalog}^{CV}$ BCQ q:* $\mathbb{P}, \mathbb{D} \models q$ *iff* $\mathsf{tr}(\mathbb{P}), \mathsf{tr}(\mathbb{D}) \models \mathsf{tr}(q)$,
2. *for $k = \mathsf{s\text{-}height}(\mathbb{P})$, every standard chase sequence for $\mathsf{tr}(\mathbb{P})$ over $\mathsf{tr}(\mathbb{D})$ terminates in a number of steps that is k-exponential in the size of $\mathbb{D}$ and $(k+2)$-exponential in the size of $\mathbb{P}$. Assuming bounded tuple height, it is $(k+1)$-exponential in the size of $\mathbb{P}$.*

Proof sketch. The correctness of the transformation has been discussed before. For the complexity, we estimate the maximal number of complex terms that may be required. For a set height k, we find that this is in $O(2^{2^{\cdot^{\cdot^{2^{a^t}}}}})$, where there are k occurrences of 2, a is the maximal arity of tuples, and t is the tuple height. Our simulation admits up to $m!$ many representations of each set of size m, corresponding to the order of adding elements, which is still exponential in a polynomial over m since $m! < m^m = 2^{m \log m}$. The number of facts and rule applications in the chase for $\mathsf{tr}(\mathbb{P})$ over $\mathsf{tr}(\mathbb{D})$ is polynomial in the resulting $(k+2)$-exponential number of terms, since every fact is completely determined by a fixed number of complex terms (defined by our translation rather than by the input). ◀

Although this matches the lower complexity bounds of Theorem 6, the above translation is not efficient, already since the rules in Figure 2 create all possible terms over the given sorts. This can instead be done "on demand" if we add additional premises, e.g., $\mathsf{mkSU}(x, V)$ in rule (22). For any rule $\varphi \to \psi$ with a body atom $\mathsf{SU}(x, V, W) \in \varphi$, we can then create an additional rule $\varphi \setminus \{\mathsf{SU}(x, V, W)\} \to \mathsf{mkSU}(x, V)$ to trigger the computation of a relevant fact for $\mathsf{SU}(x, V, W)$. If several auxiliary predicates occur, they are ordered and computed successively, in the manner of the *magic sets* transformation [2].

5 A Tractable Fragment

We now turn our attention to a fragment of $\mathsf{Datalog}^{\mathrm{CV}}$ that still supports sets but keeps reasoning tractable. To this end, we require that the cardinality of any set derived by the program on any database remains bounded by a fixed k. Although it is generally undecidable whether a program has this property, we develop sufficient conditions for bounded cardinality.

▶ **Definition 11.** *A* $\mathsf{Datalog}^{CV}$ *ground fact* α *has* k-bounded cardinality *if all set terms occurring in* α *are of the form* $\{t_1, \dots, t_n\}$ *with* $n \leq k$. *A* $\mathsf{Datalog}^{CV}$ *program* $\mathbb{P}$ *has* k-bounded cardinality *if, for any database* $\mathbb{D}$, *all ground facts* α *with* $\mathbb{P}, \mathbb{D} \models \alpha$ *have* k*-bounded cardinality.* $\mathbb{P}$ *has* bounded cardinality *if it has* k*-bounded cardinality for some* $k \in \mathbb{N}$.

Note that k-bounded cardinality implies k'-bounded cardinality for all $k' \geq k$. Also note that programs containing unsafe set variables generally do not have bounded cardinality, since such variables range over all possible sets over the active domain.

▶ **Example 12.** The following program $\mathbb{P}$ has 2-bounded cardinality.

$$\mathsf{e}(x) \to \mathsf{s}(\{x\}) \tag{39}$$

$$\mathsf{s}(X) \wedge \mathsf{s}(Y) \to \mathsf{p}(X \cup Y) \tag{40}$$

We obtain a program $\mathbb{P}'$ that does not have bounded cardinality if we replace (40) by (41).

$$\mathsf{s}(X) \wedge \mathsf{s}(Y) \to \mathsf{s}(X \cup Y) \tag{41}$$

Indeed, s is now closed under unions, hence it contains a set whose cardinality is the number of $e(x)$ facts, which depends on the size of the database.

▶ **Theorem 13.** *Every* k*-bounded cardinality* $\mathsf{Datalog}^{CV}$ *program* $\mathbb{P}$ *and database* $\mathbb{D}$ *for* $\mathbb{P}$ *can be translated into a Datalog program* $\mathsf{dl}(\mathbb{P})$ *and a database* $\mathsf{dl}(\mathbb{D})$ *such that:*

1. *for every ground* $\mathsf{Datalog}^{CV}$ *fact* α*:* $\mathbb{P}, \mathbb{D} \models \alpha$ *iff* $\mathsf{dl}(\mathbb{P}), \mathsf{dl}(\mathbb{D}) \models \mathsf{dl}(\alpha)$,
2. *the translations* $\mathsf{dl}(\cdot)$ *are* PTime*-computable with respect to* k *and the size of the input and* ExpTime*-computable with respect to the height of the schema of* $\mathbb{P}$.

Proof. Let $\mathbb{P}$ be a k-bounded cardinality $\mathsf{Datalog}^{\mathrm{CV}}$ program. The translation eliminates complex sorts by increasing the arity of predicates: we replace every position of a set sort $\tau = \{\sigma\}$ by k positions of sort σ, filling up the remaining positions with $\sqcup_\tau$ to represent sets of cardinality less than k, and every position of some tuple sort $\pi = \langle \tau_1, \tau_2, \ldots, \tau_\ell \rangle$ is replaced by ℓ positions of sorts $\tau_1, \tau_2, \ldots, \tau_\ell$, respectively. Repeating this process leads to predicates in which every position is of the domain sort. Tuple terms $\langle t_1, t_2, \ldots, t_\ell \rangle$ are easily translated, as each individual term t_i is simply placed in one of the new positions. Basic set terms are translated similarly, but for compound set terms, the situation is a bit more complicated: Similarly to the translation into TGDs in Section 4, we add $3k$-ary atoms $\mathsf{U}_\tau(\boldsymbol{t_1}, \boldsymbol{t_2}, \boldsymbol{t})$ and $\mathsf{I}_\tau(\boldsymbol{t_1}, \boldsymbol{t_2}, \boldsymbol{t})$ to rules containing $t_1 \cup_\tau t_2$ and $t_1 \cap_\tau t_2$, respectively, where the individual terms are replaced by k-tuples of terms; and use the variables $\boldsymbol{t}$ in the positions corresponding to the original term. Analogously, terms $\boldsymbol{\mathcal{P}}(t)$ are replaced by $2k$-ary facts $\mathsf{PS}_\sigma(\boldsymbol{t}, \boldsymbol{p})$, where $\boldsymbol{p}$ is the k-tuple of variables corresponding to the evaluation of the term. We can then simulate the behaviour of these set operations using further rules. Since all sets have k-bounded cardinality, no more than k of the first $2k$ positions in U_τ can be distinct from $\sqcup_\tau$, and no more than $\lfloor \log_2 k \rfloor$ of the first k positions in PS_σ can be distinct from $\sqcup_\sigma$, respectively, whereas all of the first $2k$ positions of I_τ may be distinct from $\sqcup_\tau$. In either case, however, the number of rules needed to simulate these operations depends polynomially on k, and is exponential in the height of the schema of $\mathbb{P}$. Applying these transformations to $\mathbb{P}$, $\mathbb{D}$, and q, respectively, we thus obtain $\mathsf{dl}(\mathbb{P})$, $\mathsf{dl}(\mathbb{D})$, and $\mathsf{dl}(q)$, in which all positions are of the domain sort, i.e., $\mathsf{dl}(\mathbb{P})$ is a Datalog program. ◀

If we consider schemas of unbounded height, then the arity of the required Datalog predicates may grow exponentially, though it remains bounded in terms of the database size.

▶ **Corollary 14.** *Every bounded cardinality* $\mathsf{Datalog}^{CV}$ *program has* PTIME*-complete data complexity and* 2EXPTIME*-complete combined complexity.*

Proof sketch. Upper bounds follow from Theorem 13. PTIME-hardness is inherited from plain Datalog. For the 2EXPTIME-hardness, it suffices to consider a signature without sets. By ordering (nested) tuples, we can construct a doubly-exponentially long chain as in the proof of Theorem 6. Simulating a Turing machine on this chain is again standard. ◀

The next two results establish that bounded cardinality is undecidable in general but becomes decidable for a fixed bound k.

▶ **Theorem 15.** *It is undecidable whether a* $\mathsf{Datalog}^{CV}$ *program has bounded cardinality.*

Proof. Let $\mathcal{M}$ be a deterministic Turing machine. We construct a $\mathsf{Datalog}^{\mathrm{CV}}$ program $\mathbb{P}$ such that $\mathbb{P}$ has bounded cardinality iff $\mathcal{M}$ halts on the empty tape. We consider a schema consisting of a unary predicate first and a binary predicate next. We intend databases to encode initial segments of a linear order using these predicates, but we need to ensure that $\mathbb{P}$ does not produce arbitrarily large sets on invalid encodings in a database. Hence, we collect all predecessors of an element in a set: this allows us to detect cycles and untangles elements with multiple predecessors. The next two rules realise this ($\mathsf{stepped}$ is explained below):

$$\mathsf{first}(x) \to \mathsf{step}(\emptyset, \{x\}) \wedge \mathsf{lift}(x, \{x\}) \tag{42}$$

$$\mathsf{next}(x, y) \wedge \mathsf{lift}(x, X) \wedge y \notin X \wedge \mathsf{stepped}(X) \to \mathsf{step}(X, X \cup \{y\}) \wedge \mathsf{lift}(y, X \cup \{y\}) \tag{43}$$

To axiomatise $y \notin X$ without enforcing unbounded cardinality, we use rule (12), but replace rule (10) by $\to \emptyset \subseteq \emptyset$, which is the only $\subseteq$-entailment needed for (12).

In addition, $\mathbb{P}$ contains rules that simulate $\mathcal{M}$ on the empty tape, constructing a suitable grid using step-facts, starting at $\emptyset$. These rules do not require any further sets besides those used as grid points. Once a valid transition of $\mathcal{M}$ has been performed, the used set is marked as $\mathsf{stepped}$, which allows $\mathbb{P}$ to extend the linear order by one further step. If the input contains cycles, rule (43) becomes inapplicable, as $y \notin X$ will not hold. It might happen that multiple linear orders are derived, e.g., when an element has more than one next-successors. In this case, $\mathbb{P}$ merely performs multiple simulations in parallel. If $\mathcal{M}$ halts on the empty tape after k steps, then, for every simulation, there will only be k facts of the form $\mathsf{stepped}(X)$. Each of these sets X contains at most $k-1$ elements, so $\mathbb{P}$ has bounded cardinality. If, however, $\mathcal{M}$ does not halt on the empty tape, the databases $\mathbb{D}_i = \{\mathsf{first}(0), \mathsf{next}(0,1), \dots, \mathsf{next}(i-1,i)\}$ for $i \geq 0$ witness that $\mathbb{P}$ has unbounded cardinality. ◀

▶ **Theorem 16.** *Let $k \in \mathbb{N}$ and let $\mathbb{P}$ be a $\mathsf{Datalog}^{CV}$ program. Deciding if $\mathbb{P}$ has k-bounded cardinality is* 2ExpTime*-complete, and* ExpTime*-complete for schemas of bounded height.*

Proof. Let $\star_0, \dots, \star_k$ be distinct constants not occurring in $\mathbb{P}$. The *k-critial instance* $\mathbb{D}_k$ has, for every EDB predicate $p \in \mathbf{P}^{\text{EDB}}$, all possible facts constructed using $\star_0, \dots, \star_k$: if p has sort Δ, then $\mathbb{D}_k$ contains facts $p(\star_0), \dots, p(\star_k)$; if p has sort $\langle \Delta_1, \dots, \Delta_\ell \rangle$, then $\mathbb{D}_k$ contains all facts $p(\star_{i_1}, \dots, \star_{i_\ell})$ with $i_1, \dots, i_\ell \in \{1, \dots, k\}$.

Since entailment for $\mathsf{Datalog}^{\text{CV}}$ is monotonic (i.e., adding facts to a database only ever leads to more entailed facts), any fact entailed by $\mathbb{P}$ over a database containing $k+1$ distinct constants is also entailed over $\mathbb{D}_k$.

If all facts entailed by $\mathbb{P}$ over $\mathbb{D}_k$ are k-bounded, then $\mathbb{P}$ has k-bounded cardinality, otherwise we find at least one non-k-bounded fact witnessing that $\mathbb{P}$ does not have k-bounded cardinality. To compute these facts, we can use the procedure from Theorem 4.

The computation can be stopped as soon as a fact with a set of cardinality $> k$ is inferred. Hence, for analysing complexity, we only need to consider inferences of facts with smaller sets. We can estimate the number of terms that respect this restriction by viewing each term as a tree of depth bounded by the height h of the schema, and branching factor f bounded by the maximum of k and the arity of any tuple sort. Over $k+1$ constants in $\mathbb{D}_k$, there can be at most $(k+1)^{f^h}$ such terms. This translates into a double exponential bound for the number of derivable facts, and an according complexity bound as claimed. If the height of the schema is bounded, h can be considered constant and the same calculation leads to a single exponential bound (depending on f). ◀

Derivations of "large" sets involve either recursive rule applications to construct them from smaller sets, or unsafe set variables. Thus, the absence of either feature is a sufficient condition for bounded cardinality. This is similar to how *acyclicity notions* syntactically ensure chase termination on all input databases for sets of TGDs by restricting the propagation of nulls [15]. Indeed, we adapt Weak Acyclicity [18, 19] into a criterion for bounded cardinality.

▶ **Definition 17.** *We encode* positions *in sorts and terms as lists of natural numbers, where ε is the empty list and $\cdot$ is concatenation (which we generalise to sets of lists in the usual way). The set of positions* $\mathrm{Pos}(\tau)$ *of a sort τ, is defined recursively as follows:*

- $\mathrm{Pos}(\Delta) \coloneqq \{\varepsilon\}$*;*
- $\mathrm{Pos}(\{\tau'\}) \coloneqq \{\varepsilon\} \cup (1 \cdot \mathrm{Pos}(\tau'))$*; and*
- $\mathrm{Pos}(\langle \tau_1, \dots, \tau_\ell \rangle) \coloneqq \{\varepsilon\} \cup \bigcup_{i=1}^{\ell} (i \cdot \mathrm{Pos}(\tau_i))$.

Given a term t of sort τ, the subterms $\mathrm{At}(t, w)$ at *a position $w \in \mathrm{Pos}(\tau)$ are defined recursively as follows:*

- $\mathrm{At}(t, \varepsilon) \coloneqq t$;
- $\mathrm{At}(\{t_1, \ldots, t_n\}, 1 \cdot v) \coloneqq \bigcup_{i=1}^{n} \mathrm{At}(t_i, v)$;
- $\mathrm{At}(\langle t_1, \ldots, t_\ell \rangle, i \cdot v) \coloneqq \mathrm{At}(t_i, v)$;
- $\mathrm{At}(t_1 \cup t_2, v) \coloneqq \mathrm{At}(t_1, v) \cup \mathrm{At}(t_2, v)$; *and*
- $\mathrm{At}(t_1 \cap t_2, v) \coloneqq \mathrm{At}(t_1, v) \cup \mathrm{At}(t_2, v)$,

where v is a (possibly empty) list of natural numbers. Note that $\mathrm{At}(\{\}, v \cdot 1) = \emptyset$.

All terms in $\mathrm{At}(t, w)$ *are of the same sort, which we call the* sort of t at position w. *A variable x* occurs at *position w in term t of sort τ if the sort of x is the sort of t at position w, and x occurs in a term in* $\mathrm{At}(t, w)$.

A predicate position *is a list $p \cdot w$ for a position $w \in \mathrm{Pos}(\mathsf{sort}(p))$; we denote by* $\mathrm{Pos}(p)$ *the set of all predicate positions for p. A variable x* occurs *at a predicate position $p \cdot w$ in a formula φ if φ contains an atom $p(t)$ and x occurs at position w in t.*

▶ **Example 18.** Consider the term $t = \langle a, \{\{x\}, S \cup \{y\}\}\rangle$ of sort $\tau = \langle \Delta, \{\{\Delta\}\}\rangle$. The positions of τ are $\mathrm{Pos}(\tau) = \{\varepsilon, 1, 2, 2 \cdot 1, 2 \cdot 1 \cdot 1\}$, and we have $\mathrm{At}(t, 1) = \{a\}$, $\mathrm{At}(t, 2) = \{\{\{x\}, S \cup \{y\}\}\}$, and $\mathrm{At}(t, 2 \cdot 1) = \{\{x\}, S \cup \{y\}\}$. We find that S occurs at $2 \cdot 1$ whereas x and y occur at $2 \cdot 1 \cdot 1$.

▶ **Definition 19.** *Consider a* $\mathsf{Datalog}^{CV}$ *program $\mathbb{P}$. The* WSA graph $G(\mathbb{P})$ *of $\mathbb{P}$ is a directed graph with two kinds of edges ("normal" and "special"). The vertices of $G(\mathbb{P})$ are all predicate positions of predicates in $\mathbb{P}$. $G(\mathbb{P})$ contains the following edges:*

1. *For every rule $\varphi \to \psi \in \mathbb{P}$, and every variable x that occurs in a body atom $q(s) \in \varphi$ at position $q \cdot v$ and in a head atom $p(t) \in \psi$ at position $p \cdot w$, there is an edge $q \cdot v \to p \cdot w$; it is* special *if x occurs in a subterm of the form $S \cup T$ in* $\mathrm{At}(t, w)$, *and* normal *otherwise.*
2. *For every rule $\varphi \to \psi \in \mathbb{P}$, and every variable x of set sort $\{\tau\}$ or tuple sort $\langle \tau_1, \ldots, \tau_\ell \rangle$, where a set sort occurs (directly or transitively) in a component sort τ_i, that occurs in a head atom $p(t) \in \psi$ at position $p \cdot w$ and that does not occur in the body φ, there is a special edge $p \cdot w \to p \cdot w$.*
3. *If there is an edge $q \cdot v \to p \cdot w$, and if the sort τ at position $q \cdot v$ (which is the same as the sort at position $p \cdot w$) has a position $u \in \mathrm{Pos}(\tau)$, then there is a normal edge $q \cdot v \cdot u \to p \cdot w \cdot u$.*

$\mathbb{P}$ is weakly set-acyclic *if $G(\mathbb{P})$ does not contain a directed cycle that involves a special edge.*

Intuitively, edges of the first kind model the propagation of values by rule applications. Edges of the second kind correspond to unsafe variables ranging over values containing sets. Such values always include sets that contain all constants occurring in the database. Thus, a program containing such a rule generally does not have bounded cardinality, and we thus always force a cycle in the WSA graph. Lastly, edges of the third kind model the propagation of values inside values of composite sorts. Indeed, weak set-acyclicity is easy to check:

▶ **Lemma 20.** *Deciding if $\mathbb{P}$ is weakly set-acyclic is* NL*-complete.*

Proof sketch. The WSA graph can be constructed by a logspace transducer, since the presence of any edge can be decided in L. Checking for cycles in a logspace-computable directed graph is NL-complete. ◀

We also note that any sufficiently large set produced by some program involves either an unsafe set variable, or the recursive application of some rule:

▶ **Lemma 21.** *For a* $\mathsf{Datalog}^{CV}$ *program $\mathbb{P}$ without unsafe set variables, there is $n \in \mathbb{N}$ such that a minimal chase sequence deriving a set S with $|S| \geq n$ has some rule ρ applied to a match containing a set T, where ρ is again part of the subsequence deriving T.*

Proof. Let $\rho \in \mathbb{P}$ be a rule and consider a match h for ρ. There is a polynomial $p(x)$ such that any set obtained by applying ρ yields a set of cardinality at most $p(x)$, provided that no set in h has cardinality exceeding x. Let $q(x)$ be a simultaneous upper bound for all such polynomials for rules in $\mathbb{P}$, consider the $|\mathbb{P}|$-fold composition $q^{|\mathbb{P}|} = q(\cdots(q(x))\cdots)$, and define $n := q^{|\mathbb{P}|}(1) + 1$.

If there is no database $\mathbb{D}$ such that $\mathbb{P}$ derives a set S of cardinality $|S| \geq n$ on $\mathbb{D}$, then $\mathbb{P}$ has n-bounded cardinality, and the claim is vacuously true. Otherwise, let S be such a set and $\mathbb{D}$ be such a database. Consider a minimal chase sequence deriving S. Any chase sequence without duplicate rules has at most $\mathbb{P}$ steps, and since $\mathbb{D}$ does not contain sets, such a sequence derives a set of cardinality at most $q^{|P|}(1) < n$. Thus, the minimal chase sequence deriving S must contain some rule ρ occurring in steps i and $j > i$, where step i derives a set T with $|T| < |S|$. ◀

We therefore find weak set-acyclicity a sufficient condition for bounded cardinality:

▶ **Theorem 22.** *If $\mathbb{P}$ is weakly set-acyclic, then $\mathbb{P}$ has bounded cardinality.*

Proof. Towards a contradiction, assume that $\mathbb{P}$ has unbounded cardinality, but $G(\mathbb{P})$ does not contain a directed cycle traversing a special edge. Let n be the constant from Lemma 21. Since $\mathbb{P}$ has unbounded cardinality, there is a set S with $|S| \geq n$ that $\mathbb{P}$ derives, consider a minimal chase sequence for some fact containing it. Note that any rule application in this sequence corresponds to at least one edge in $G(\mathbb{P})$. By Lemma 21, there is a set T derived by a rule ρ that is propagated to some fact partaking in another match for ρ. Thus, ρ is part of a directed cycle in $G(\mathbb{P})$. Furthermore $|T| < |S|$, since the chase sequence is minimal. Thus one of the rule applications corresponding to this cycle must involve a union in the head of the rule, making the corresponding edge in $G(\mathbb{P})$ special. ◀

Weakly set-acyclic programs constitute a tractable fragment of $\mathsf{Datalog}^{\mathrm{CV}}$ for which membership is decidable. But even simple bounded-cardinality programs may not be WSA:

▶ **Example 23.** Consider again the programs $\mathbb{P}$ and $\mathbb{P}'$ from Example 12. The WSA graph of $\mathbb{P}$ has edges $e \cdot \epsilon \to s \cdot 1$, $s \cdot \epsilon \to p \cdot \epsilon$, and $s \cdot 1 \to p \cdot 1$. Therefore, $\mathbb{P}$ is WSA. The WSA graph for $\mathbb{P}'$ contains the special edge $s \cdot \epsilon \to s \cdot \epsilon$, and $\mathbb{P}'$ is not WSA. Let $\bar{\mathbb{P}}$ consist of $\mathbb{P}$ and (44):

$$s(X) \wedge s(Y) \wedge p(S) \to s(S \cap (X \cup Y)) \tag{44}$$

The WSA graph of $\bar{\mathbb{P}}$ has the special edge $s \cdot \epsilon \to s \cdot \epsilon$, but $\bar{\mathbb{P}}$ has 2-bounded cardinality.

We can improve over WSA by estimating the maximal cardinality of sets produced by a rule more cautiously. We therefore construct a system of inequalities that correspond to lower bounds on the cardinalities of sets. These bounds are expressions built from natural numbers and the operators $+$, min, and max. Then the sum of these lower bounds has a minimum if and only if the program has bounded cardinality.

▶ **Definition 24.** *Let $\mathbb{P}$ be a $\mathsf{Datalog}^{CV}$ program. For a set sort $\tau = \{\tau'\}$, let P_τ be the set of all predicate positions of sort τ. A predicate position $p \cdot w$ in P_τ is a* target position *if p occurs in the head of some rule $\rho \in \mathbb{P}$. With every predicate position $p \cdot w \in P_\tau$, associate a variable $x_{p \cdot w}$. Set $X := \bigcup_{\tau = \{\tau'\}} \{x_{p \cdot w} \mid p \cdot w \in P_\tau\}$, and let $T \subseteq X$ be the set of all target variables. Recursively define the* lower bound $[\![t]\!]^\rho_{p \cdot w}$ *of a term t occurring at $p \cdot w$ in rule ρ:*

1. *if $t \in \mathbf{V}$ is a variable, let* $\mathrm{bpos}(t)$ *be the set of all predicate positions of* $\mathrm{body}\,\rho$ *that t occurs at and set $[\![t]\!]^\rho_{p \cdot w} := \max\{x_{q \cdot v} \mid q \cdot v \in \mathrm{bpos}(t)\}$;*
2. *$[\![\{t_1, \ldots, t_\ell\}]\!]^\rho_{p \cdot w} := \ell$;*

3. $[\![(t_1 \cap t_2)]\!]^{\rho}_{p\cdot w} \coloneqq \min([\![t_1]\!]^{\rho}_{p\cdot w}, [\![t_2]\!]^{\rho}_{p\cdot w})$; *and*
4. $[\![(t_1 \cup t_2)]\!]^{\rho}_{p\cdot w} \coloneqq [\![t_1]\!]^{\rho}_{p\cdot w} + [\![t_2]\!]^{\rho}_{p\cdot w}$.

Every variable $x \in X$ has the associated inequality $x \geq 0$, every target variable $x_{p\cdot w}$ has associated inequalities for terms t occurring at $p \cdot w$ in the head of rule ρ: $x_{p\cdot w} \geq [\![t]\!]^{\rho}_{p\cdot w}$. The system of cardinality constraints $\mathsf{card}(\mathbb{P})$ *is the system of all such inequalities. The* cardinality constraints problem *is minimising $\sum_{x\in T} x$ subject to* $\mathsf{card}(\mathbb{P})$.

Intuitively speaking, we obtain a bound ℓ for set literals $\{t_1, \ldots, t_\ell\}$ by assuming that all terms are distinct. For intersections $t_1 \cap t_2$, the bound is the minimum of the bounds for t_1 and t_2, whereas for unions, it is the sum of the bounds (assuming that, in the worst case, the sets are disjoint). For variables, we simply take the maximum over all bounds for the same sort that occur in the body of the rule.

The cardinality constraints problem allows us to strengthen the bound from Lemma 21:

▶ **Lemma 25.** *Let $\mathbb{P}$ be a* $\mathsf{Datalog}^{CV}$ *program. If the cardinality constraints problem has optimal value k, any minimal chase sequence deriving a set S with $|S| \geq k+1$ has some rule ρ applied to a match containing a set T, where ρ is again part of the subsequence deriving T.*

▶ **Theorem 26.** *If the cardinality constraints problem for some* $\mathsf{Datalog}^{CV}$ *program $\mathbb{P}$ has an optimal solution, then $\mathbb{P}$ has bounded cardinality. Deciding whether this is the case can be done in polynomial time with respect to the size of $\mathbb{P}$.*

Proof. Note that the cardinality constraints problem for $\mathbb{P}$ is always *bounded*, i.e., it always has only finitely many possible solutions, since we require $x \geq 0$ for all variables $x \in X$. Thus, it has an optimal solution precisely when it has at least one solution, i.e., when it is *feasible*. Assume for a contradiction that $\mathbb{P}$ has unbounded cardinality, but that the cardinality constraints problem has optimal value k. By Lemma 25, there is a set S with $|S| \geq k+1$ such that any minimal chase sequence applies rule ρ to some set T with ρ part of the subsequence deriving T. Without loss of generality, assume that both S and T occur at the same predicate position $p \cdot w$ (since $\mathbb{P}$ has unbounded cardinality, we can chose S large enough) in steps i and $j > i$. Let $p \cdot w = q^i \cdot v^i, q^{i+1} \cdot v^{i+1}, \ldots, q^{j-1} \cdot v^{j-1}, q^j \cdot v^j = p \cdot w$ be the positions along the subsequence. We have $x_{p\cdot w} \geq x_{q^{i+1}\cdot v^{i+1}} \geq \cdots \geq x_{q^{j-1}\cdot v^{j-1}} \geq x_{p\cdot w}$. Since $|S| > |T|$, one of the rules applied in the subsequence involves a union. Thus one of the inequalities is strict and the cardinality constraints problem is infeasible, which is the desired contradiction.

Since $\mathbb{P}$ only has polynomially many positions, the cardinality constraints problem for $\mathbb{P}$ is of polynomial size with respect to $\mathbb{P}$. By using additional variables, $\mathsf{card}(\mathbb{P})$ can be transformed into an equifeasible linear program (i.e., a system of linear inequalities that admits an optimal solution precisely when $\mathsf{card}(\mathbb{P})$ does): inequalities of the form $x \geq \max\{t_1, \ldots, t_\ell\}$ are replaced by ℓ inequalities $x \geq t_1, \ldots, x \geq t_\ell$, and inequalities of the form $x \geq \min(t_1, t_2)$ are replaced by $x = t_1 - y$, $y \geq 0$, and $y \geq t_1 - t_2$, where y is a fresh variable. The resulting linear program is still of polynomial size, and solving such linear programs can be done in polynomial time [24]. ◀

The system of cardinality constraints is not a sufficient condition for bounded cardinality either, but it can capture bounded cardinality for programs that are not weakly set-acyclic:

▶ **Example 27.** Consider again the program $\bar{\mathbb{P}}$ from Example 23. Then $\mathsf{card}(\bar{\mathbb{P}})$ consists of the following inequalities with optimal solution $x_{s\cdot\epsilon} = 2 = x_{p\cdot\epsilon}$ and optimal value $k = 4$:

$$x_{s\cdot\epsilon} \geq 0 \qquad x_{p\cdot\epsilon} \geq 0$$
$$x_{s\cdot\epsilon} \geq 1 \qquad x_{p\cdot\epsilon} \geq 2 \qquad x_{s\cdot\epsilon} \geq \min(x_{p\cdot\epsilon}, x_{s\cdot\epsilon} + x_{s\cdot\epsilon})$$

6 Linear Datalog$^{\text{CV}}$

For both Datalog and TGDs, linearity is a syntactic criterion that corresponds to a fragment with lower complexities [21, 20]. It thus seems prudent to investigate whether the analogous fragment of Datalog$^{\text{CV}}$ enjoys similarly reduced complexities.

▶ **Definition 28.** *A* Datalog^{CV} *rule* $\varphi \to \psi$ *is a* linear Datalog^{CV} *rule if both* φ *and* ψ *each comprise exactly one atom. A* Datalog^{CV} *program* $\mathbb{P}$ *is linear if all rules* $\rho \in \mathbb{P}$ *are linear.*

Unlike for Datalog$^{\text{CV}}$, we allow databases for linear Datalog$^{\text{CV}}$ programs to contain facts of arbitrary sorts, as such facts cannot generally be constructed using linear rules. We find that with one added layer of nested sets, deciding entailment for linear Datalog$^{\text{CV}}$ is just as hard as for Datalog$^{\text{CV}}$. In other words, for the same set height, the complexity of linear Datalog$^{\text{CV}}$ is indeed at most one exponent lower than for non-linear Datalog$^{\text{CV}}$.

▶ **Theorem 29.** *Let* $\mathbf{S}$ *be a schema with* $k = \mathsf{s\text{-}height}(\mathbf{S}) > 0$*, let* $\mathbb{P}$ *be a linear* Datalog^{CV} *program and* $\mathbb{D}$ *a database over* $\mathbf{S}$*, and let* α *be a ground fact. Deciding* $\mathbb{P}, \mathbb{D} \models \alpha$ *is* $(k-1)$ExpTime*-hard for data complexity and* $(k+1)$ExpTime*-hard for combined complexity. If* $\mathsf{t\text{-}height}(\mathbf{S})$ *is bounded, the combined complexity drops to* kExpTime*-hard.*

Proof sketch. The proof follows the idea from the proof of Theorem 6. We use a single predicate facts holding tuples of sets, with each component corresponding to one of the original predicates: Let $p_1, p_2, \dots$ be a fixed enumeration of all predicates occurring in $\mathbb{P}$ with sorts $\tau_1, \tau_2, \dots$. Then facts has sort $\langle \{\tau_1\}, \{\tau_2\}, \dots \rangle$, and component i corresponds to facts for p_i. For a formula Φ, define $\mathrm{ts}_\Phi(p) := \{\boldsymbol{t}\}$, where $\boldsymbol{t}$ is the (possibly empty) list of terms t for which $p(t)$ occurs in Φ. We then translate a rule $\rho = \varphi \to \psi$ into (45), and translate $\mathbb{D}$ as $\mathsf{dbFacts}(\mathrm{ts}_{\mathbb{D}}(p_1), \mathrm{ts}_{\mathbb{D}}(p_2), \dots)$, where we view $\mathbb{D}$ as a conjunction of atoms.

$$\mathsf{facts}(y_1 \cup \mathrm{ts}_\varphi(p_1), y_2 \cup \mathrm{ts}_\varphi(p_2), \dots) \to \mathsf{facts}(y_1 \cup \mathrm{ts}_\rho(p_1), y_2 \cup \mathrm{ts}_\rho(p_2), \dots) \tag{45}$$

Lastly, the linear rule $\mathsf{dbFacts}(\boldsymbol{y}) \to \mathsf{facts}(\boldsymbol{y})$ derives the initial facts from $\mathbb{D}$. ◀

Note that, even with this encoding, negation cannot be simulated by $\notin$, since we cannot require rules to be applied only after all facts for some predicate p_i have been derived.

7 Discussion and Future Work

We have formalised Datalog$^{\text{CV}}$, a positive extension of Datalog with complex values, identified its complexity, and developed a translation into terminating TGD sets. In sharp contrast to Relationlog [28], which has the same complexity as Datalog, Datalog$^{\text{CV}}$ can express highly complex queries, and unlike Set-extended Datalog [39], it supports tuples as well as sets.

We have shown that bounded cardinality programs form a fragment with tractable reasoning. Since it is undecidable whether a program has bounded cardinality, we have proposed two decidable sufficient conditions for bounded cardinality: weak set-acyclicity and the cardinality constraints problem.

Regarding future works, our translations put complex value reasoning in reach of rule engines such as VLog [11] and RDFox [31], which have the potential of addressing many of the applications from the introduction. On the theoretical side, *(stratified) negation* would be a natural extension to study, also regarding its utility for expressing query languages such as MARPL [30] and eMARPL [29]. Moreover, while it was recently shown that chase-terminating tuple-generating dependencies capture all decidable monotonic queries [8], the expressive power of Datalog$^{\text{CV}}$ and its tractable fragments remains open.

References

1 Serge Abiteboul and Stéphane Grumbach. COL: A logic-based language for complex objects. In Joachim W. Schmidt, Stefano Ceri, and Michele Missikoff, editors, *Proc. 1st Int. Conf. on Extending Database Technology (EDBT'88)*, volume 303 of *LNCS*, pages 271–293. Springer, 1988.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1994.

3 Serge Abiteboul and Paris C. Kanellakis. Object identity as a query language primitive. *J. ACM*, 45(5):798–842, 1998.

4 Shqiponja Ahmetaj, Magdalena Ortiz, and Mantas Simkus. Rewriting guarded existential rules into small datalog programs. In Benny Kimelfeld and Yael Amsterdamer, editors, *Proc. 21st Int. Conf. on Database Theory (ICDT'18)*, volume 98 of *LIPIcs*, pages 4:1–4:24. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

5 Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutiérrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proc. 2018 ACM SIGMOD Int. Conf. on Management of Data*, pages 1421–1432. ACM, 2018.

6 Catriel Beeri, Shamim A. Naqvi, Raghu Ramakrishnan, Oded Shmueli, and Shalom Tsur. Sets and negation in a logic database language (LDL1). In Moshe Y. Vardi, editor, *Proc. 6th Symposium on Principles of Database Systems (PODS'87)*, pages 21–37. ACM, 1987.

7 Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. of the ACM*, 31(4):718–741, 1984.

8 Camille Bourgaux, David Carral, Markus Krötzsch, Sebastian Rudolph, and Michaël Thomazo. Capturing homomorphism-closed decidable queries with existential rules. In Meghyn Bienvenu, Gerhard Lakemeyer, and Esra Erdem, editors, *Proc. 18th International Conference on Principles of Knowledge Representation and Reasoning (KR'21)*, pages 141–150, 2021.

9 Pierre Bourhis, Juan L. Reutter, and Domagoj Vrgoc. JSON: data model and query languages. *Inf. Syst.*, 89:101478, 2020.

10 Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Finitely recursive programs: Decidability and bottom-up computation. *J. of AI Commun.*, 24(4):311–334, 2011.

11 David Carral, Irina Dragoste, Larry González, Ceriel J. H. Jacobs, Markus Krötzsch, and Jacopo Urbani. Vlog: A rule engine for knowledge graphs. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *Proc. 18th Int. Semantic Web Conf. (ISWC'19), Part II*, volume 11779 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2019.

12 David Carral, Irina Dragoste, and Markus Krötzsch. Reasoner = logical calculus + rule engine. *KI*, 2020.

13 David Carral, Irina Dragoste, Markus Krötzsch, and Christian Lewe. Chasing sets: How to use existential rules for expressive reasoning. In Sarit Kraus, editor, *Proc. 28th Int. Joint Conf. on Artificial Intelligence (IJCAI'19)*. ijcai.org, 2019.

14 Angelos Charalambidis, Christos Nomikos, and Panos Rondogiannis. The expressive power of higher-order datalog. *Theory Pract. Log. Program.*, 19(5-6):925–940, 2019.

15 Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *J. of Artificial Intelligence Research*, 47:741–808, 2013.

16 Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

17 Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In Maurizio Lenzerini and Domenico Lembo, editors, *Proc. 27th Symposium on Principles of Database Systems (PODS'08)*, pages 149–158. ACM, 2008.

18 Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Proc. 9th Int. Conf. on Database Theory (ICDT'03)*, volume 2572 of *LNCS*, pages 225–241. Springer, 2003.

19 Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

20 Georg Gottlob, Marco Manna, and Andreas Pieris. Polynomial rewritings for linear existential rules. In Qiang Yang and Michael Wooldridge, editors, *Proc. 24th Int. Joint Conf. on Artificial Intelligence (IJCAI'15)*, pages 2992–2998. AAAI Press, 2015.

21 Georg Gottlob and Christos H. Papadimitriou. On the complexity of single-rule datalog queries. *Inf. Comput.*, 183(1):104–122, 2003.

22 Marc Gyssens, Dan Suciu, and Dirk Van Gucht. Equivalence and normal forms for the restricted and bounded fixpoint in the nested algebra. *Inf. Comput.*, 164(1):85–117, 2001.

23 Jan Hidders, Jan Paredaens, and Jan Van den Bussche. J-logic: Logical foundations for JSON querying. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proc. 36th Symposium on Principles of Database Systems (PODS'17)*, pages 137–149. ACM, 2017.

24 Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

25 Michael Kifer and James Wu. A logic for programming with complex objects. *J. of Comput. Syst. Sci.*, 47(1):77–120, 1993.

26 Markus Krötzsch, Maximilian Marx, and Sebastian Rudolph. The power of the terminating chase. In *Proc. 22nd Int. Conf. on Database Theory (ICDT'19)*, volume 127 of *LIPIcs*, pages 3:1–3:17. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2019.

27 Gabriel M. Kuper. Logic programming with sets. *J. of Comput. Syst. Sci.*, 41(1):44–64, 1990.

28 Mengchi Liu. Relationlog: A typed extension to datalog with sets and tuples. *J. of Log. Program.*, 36(3):271–299, 1998.

29 David Martin and Peter F. Patel-Schneider. Wikidata constraints on MARS. In Lucie-Aimée Kaffee, Oana Tifrea-Marciuska, Elena Simperl, and Denny Vrandečić, editors, *Proc. 1st Wikidata Workshop (Wikidata 2020)*, volume 2773 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2020.

30 Maximilian Marx, Markus Krötzsch, and Veronika Thost. Logic on MARS: Ontologies for generalised property graphs. In Carles Sierra, editor, *Proc. 26th Int. Joint Conf. on Artificial Intelligence (IJCAI'17)*, pages 1188–1194. ijcai.org, 2017.

31 Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. Rdfox: A highly-scalable RDF store. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II*, volume 9367 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2015.

32 Magdalena Ortiz, Sebastian Rudolph, and Mantas Simkus. Worst-case optimal reasoning for the Horn-DL fragments of OWL 1 and 2. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Proc. 12th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'10)*, pages 269–279. AAAI Press, 2010.

33 Jan Paredaens and Dirk Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, 1992.

34 Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.

35 Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the SAP HANA database. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pages 403–420, 2013.

36 Dan Suciu. Bounded fixpoints for complex objects. *Theor. Comput. Sci.*, 176(1-2):283–328, 1997.

37 Shalom Tsur and Carlo Zaniolo. LDL: A logic-based data language. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *Proc. 12th Int. Conf. on Very Large Data Bases (VLDB'86)*, pages 33–41. Morgan Kaufmann, 1986.

38 Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10), 2014.

39 Qing Zhou and Ligong Long. SEDatalog: A set extension of datalog. In Zhongzhi Shi and Qing He, editors, *Proc. 2nd Int. Conf. on Intelligent Information Processing (IFIP'04)*, volume 163 of *IFIP*, pages 383–388. Springer, 2004.