J-Logic: a Logic for Querying JSON

Jan Hidders
University of London

Jan Paredaens
Universiteit Antwerpen

Jan Van den Bussche
Universiteit Hasselt

Abstract

We propose a logical framework, based on Datalog, to study the foundations of querying JSON data. The main feature of our approach, which we call J-Logic, is the emphasis on paths. Paths are sequences of keys and are used to access the tree structure of nested JSON objects. J-Logic also features "packing" as a means to generate a new key from a path or subpath. J-Logic with recursion is computationally complete, but many queries can be expressed without recursion, such as deep equality. We give a necessary condition for queries to be expressible without recursion. Most of our results focus on the deterministic nature of JSON objects as partial functions from keys to values. Predicates defined by J-Logic programs may not properly describe objects, however. Nevertheless we show that every object-to-object transformation in J-Logic can be defined using only objects in intermediate results. Moreover we show that it is decidable whether a positive, nonrecursive J-Logic program always returns an object when given objects as inputs. Regarding packing, we show that packing is unnecessary if the output does not require new keys. Finally, we show the decidability of query containment for positive, nonrecursive J-Logic programs.

This paper is the extended version of an earlier version published in the proceedings of SIGMOD/PODS 2017 [23].

1 Introduction

JSON is a popular semistructured data model used in NoSQL systems and also integrated in relational systems. Proposals for expressive query languages for JSON include JSONiq [17, 18], which is based on XQuery, and SQL++ [30], which is based on SQL. Schema formalisms for JSON are also being investigated [32]. Hence the time is ripe to investigate the logical foundations of JSON querying, which is the goal of the present paper.

A JSON object is a partial function, mapping keys to values. Here, a value is either an atomic value or an object in turn. Hence, objects can be nested, and thus can be viewed as trees, similarly to XML documents. JSON trees have some special characteristics, however, which form the starting point of our work. A first difference with XML trees is that JSON trees are edge-labeled rather than node-labeled; the keys are the edge labels. More importantly, JSON trees are *deterministic* in the sense of Buneman, Deutsch and Tan [11, 34]. Specifically, since objects are functions, different edges from a common parent must have different labels.[1]

Determinism is convenient because paths starting in the root of a given tree can be identified with key sequences.[2] This suggests an alternative view of objects as sets of path–value pairs, where

---

[1]Buneman, Deutsch and Tan actually considered an extension of JSON where keys need not be atomic, but can be objects in turn.

[2]In JSON Schema [32], key sequences are called "JSON pointers".

$$T(\langle @x.@y\rangle.r.@x.\$x' : @u) \leftarrow R(@x.\$x' : @u), S(@y.\$y' : @v)$$
$$T(\langle @x.@y\rangle.s.@y.\$y' : @v) \leftarrow R(@x.\$x' : @u), S(@y.\$y' : @v)$$

Figure 1: J-Logic program defining T as the Cartesian product of R and S. Here, $@x$ and $@y$ are atomic variables, binding to the top-level atomic keys of R and S respectively; $\$x'$ and $\$y'$ are path variables, binding to the paths in the subobjects below $@x$ and $@y$ in R and S respectively. The variables $@u$ and $@v$ bind to atomic values stored in the leaves. The dot indicates concatenation. We also use constant keys r and s to indicate the R- and S-parts of each pair of the Cartesian product.

each path is a path from the root to a leaf, and the corresponding value is the atomic value of that leaf. We call such a set of path–value pairs an *object description*. Since paths are sequences of keys, we are led to the conclusion that to query JSON objects, we need a query language that can work with sets of sequences.

At the same time, the theory of query languages is solidly grounded in logic [2]. Datalog in particular is a convenient logic-based language with a long tradition in data management research and a wide variety of current applications [24, 15, 7, 20].

We are thus motivated to investigate the logical foundations for JSON querying within a Datalog language for sets of sequences. Such a language, called sequence Datalog, has already been introduced by Bonner and Mecca [9, 29, 10]. Bonner and Mecca were primarily interested in expressive sequence manipulation, of the kind needed in bioinformatics applications. They reported results on expressiveness, complexity of computations, and on ways to combine recursion with sequence concatenation while still guaranteeing termination or tractability.

In this paper, we focus more on questions motivated by JSON querying and deterministic semistructured data. Thereto, we propose a new approach based on sequence Datalog, called J-Logic. Moreover, J-Logic adds a feature for constructing new keys, called *packing*. Key generation is necessary if we want the result of a query over objects to be again an object. Consider, for example, the Cartesian product of two objects that have N keys each. The result needs to be a object with N^2 keys. So, we cannot manage by just reusing the keys from the input; new keys must be generated.

The creation of new data elements (keys, identifiers, nodes, and so on) in the result of a query has already been considered in many contexts, such as highly expressive languages [4, 5], object databases [3, 22, 28], information integration [21], data exchange [6], and ontology based data access [33]. The popular languages XQuery and SPARQL both have node creation. In logic based approaches, element creation is typically achieved through the use of Skolem functions [25, 26].

In J-Logic, however, we can take advantage of having sequences in the language. We can generate new keys simply by *packing* a key sequence s into a new key $\langle s\rangle$. For example, consider two objects

$$R = \{a : o_1, b : o_2\} \text{ and } S = \{c : o_3, d : o_4\},$$

where o_1, o_2, o_3, and o_4 are subobjects. We can represent the Cartesian product of R and S by the object

$$T = \{\langle a.c\rangle : \{r : \{a : o_1\}, s : \{c : o_3\}\},$$
$$\langle a.d\rangle : \{r : \{a : o_1\}, s : \{d : o_4\}\},$$
$$\langle b.c\rangle : \{r : \{b : o_2\}, s : \{c : o_3\}\},$$
$$\langle b.d\rangle : \{r : \{b : o_2\}, s : \{d : o_4\}\}\}.$$

The two J-Logic rules in Figure 1 accomplish this.

Packed keys should be seen as an intermediate construct. We envisage that any packed keys present in the final result of a query will be replaced by fresh identifiers, as in the ILOG approach [25]. For example, T above could be returned in the following form:

$$\begin{aligned}
T = \{ & t_1 : \{r : \{a : o_1\}, s : \{c : o_3\}\}, \\
& t_2 : \{r : \{a : o_1\}, s : \{d : o_4\}\}, \\
& t_3 : \{r : \{b : o_2\}, s : \{c : o_3\}\}, \\
& t_4 : \{r : \{b : o_2\}, s : \{d : o_4\}\}\}.
\end{aligned}$$

The aforementioned languages SQL++ and JSONiq do not have key generation: there, the Cartesian product can be computed as a bag (or sequence) of objects, but not as one object itself. Key generation can thus be seen as an alternative to adding an extra collection feature (like bags, or sequences if we agree on some way to order objects) to the query language. We admit that a bag of objects could be easily transformed into one object by generating fresh keys. Thus the two approaches (key generation, or bags that are eventually transformed into objects) are largely equivalent. In J-Logic we have chosen for key generation through packing, because it is a lightweight addition to sequence Datalog. Moreover, it allows us to work with just a single kind of collections, namely, objects (more precisely, object descriptions).

In this paper we will show the following results.

1. J-Logic programs may be recursive, but we are mostly interested in the nonrecursive case. Nonrecursive programs have polynomial-time data complexity, and due to the use of sequence variables, nonrecursive programs are already quite powerful. We give a necessary condition on queries computable by nonrecursive programs, which can serve as a tool to show that certain queries involving objects of unbounded depth require recursion. (Nonrecursive J-Logic over objects of bounded depth is essentially equivalent to relational algebra.)

2. We show the technical result that packing, while convenient and necessary in general, is not needed to compute queries from flat inputs to flat outputs. Here, *flat* means that no packed keys occur in the data. An open question is whether this can be done without recursion (our simulation of packing needs recursion). An affirmative answer would yield a result analogous to the "flat–flat theorem" for the nested relational algebra [31] or calculus [12].

3. In J-Logic, a JSON object is described as a mapping from root-to-leaf paths to atomic values. Accordingly, predicates defined by J-Logic rules are relations between paths and atomic values. Not every such relation properly describes a JSON object, however. Nevertheless, we show the "object–object theorem": every query from objects to objects, computable by a J-Logic program, is computable by a J-Logic program so that every intermediate relation is a proper object description.

4. The object–object theorem assumes a J-Logic program that maps objects to objects. But can we check this? We show that the object–object property is decidable for positive, nonrecursive programs. We do this by adapting the chase procedure for equality-generating dependencies, well known from relational databases [2]. In our model, however, the chase is not complete in general. We nevertheless can use it resolve our problem.

3

5. Finally, we show that the containment problem for positive, nonrecursive programs, over flat instances, is decidable. To the best of our knowledge, the containment problem was not yet addressed in the setting of sequence Datalog. We solve the problem in our setting by extending the known inclusion test for pattern languages over an infinite alphabet [19].

This paper is further organized as follows. In Section 2 we introduce our formalization of the JSON data model. In Section 3 we define J-Logic. In Section 4 we discuss the expressive power of nonrecursive J-Logic and state the flat–flat theorem. In Section 5 we discuss the problem of proper object descriptions, state the object–object theorem, and study the object–object decision problem. Section 6 is devoted to the containment problem. We conclude in Section 7.

2 A formal data model based on JSON

We begin by defining our formalization of the JSON data model. From the outset we assume an infinite domain **dom** of atomic data elements, which we call *atomic keys*. In practice, these would be strings, numbers, or any other type of data that the database system treats as atomic. Now the sets of *values* and *objects* are defined as the smallest sets satisfying the following:

- Every atomic key is a value;

- Every object is a value;

- Every mapping from a finite set of atomic keys to values is an object.

Recall that a mapping is a set of pairs where no two pairs have the same first component. Thus, an object is a set of key–value pairs. It is customary to write a key–value pair (k, v) in the form $k : v$. For an object o and a key a, we sometimes use the notation $o.a$ for the a-value of o, i.e., for $o(a)$.

Example 2.1. Using strings such as 'name', 'age', 'anne', 'bob' and 'chris', and numbers such as 12, 18 and 24, as atomic keys, the following are three examples of objects:

$$o_1 = \{\text{name} : \text{anne}, \text{age} : 12\}$$
$$o_2 = \{\text{name} : \text{bob}, \text{age} : 18\}$$
$$o_3 = \{\text{name} : \text{chris}, \text{age} : 24\}$$

Since objects can be nested, the following is also an object:

$$o = \{\text{name} : \text{john}, \text{children} : \{1 : o_1, 2 : o_2, 3 : o_3\}\}$$

We have $o.\text{children}.2 = o_2$. Finally, note that the set $\{\text{name} : \text{anne}, \text{name} : \text{bob}\}$ is not an object since it is not a well-defined mapping. The set $\{\text{anne} : \text{name}, \text{bob} : \text{name}\}$, however, is perfectly allowed as an object. ☐

Remark 2.2. Some remarks are in order.

1. In the JSON standard [16], the keys in an object can only be strings, but values can be numbers. In our formalization we make no distinction between different types of atomic data, which explains the example above where we used the numbers 1, 2 and 3 as keys. In the language JavaScript, an array may be viewed as an object with numbers as keys. So, our approach is not too much at odds with reality.

4

2. Indeed, the JSON standard also has arrays besides objects. In this paper we focus on unordered objects. An extension of our approach, where a total order is assumed on atomic keys (and extended to packed keys, see later) seems feasible and would be able to model arrays.

3. The term "atomic *key*" is a bit misleading, as these elements may not only be used as keys, but also as values. Indeed, that keys can occur as data values, and vice versa, is a characteristic feature of JSON.

Packed keys Until now we have defined an object as a mapping from atomic keys to values. Since these values can be objects in turn, we can use sequences of atomic keys to navigate deeper inside an object. Sequences of keys will be called paths. Moreover, we also introduce packed keys, as they can be created by J-Logic rules. Formally, the sets of *keys* and *paths* are defined as the smallest sets such that

- every atomic key is a key;

- if p is a path then $\langle p \rangle$ is a key, called a *packed key*;

- every nonempty finite sequence of keys is a path.

In our notation, we use dots to separate the elements of a sequence. At the same time, the dot will be used to denote concatenation of paths.

Example 2.3. Let a and b be atomic keys. Then $a.b$ is a path; $k = \langle a.b \rangle$ is a packed key; $p = b.b.k.a$ is again a path; and $\langle p \rangle$ is again a packed key. □

From now on we allow packed keys in objects. Thereto we generalize the notion of object by defining an object to be a mapping from a finite set of keys to values. Thus, keys need not be atomic but can also be packed. We already saw an example of a object with packed keys, T in the Introduction.

Object descriptions An object can be visualized as a tree, where edges are labeled with keys and leaves are labeled with *atomic values*: atomic keys or $\emptyset$ (the empty object). Thus, we can completely describe an object by listing all paths from the root to the leaves, and, for each such path, giving the label of the corresponding leaf.

Example 2.4. Recall the object o from Example 2.1. Figure 2 shows o as a tree. Its description as a set of path–value pairs is as follows:

$$
\begin{aligned}
&\text{name : john} \\
&\text{children.1.name : anne} \\
&\text{children.1.age : 12} \\
&\text{children.2.name : bob} \\
&\text{children.2.age : 18} \\
&\text{children.3.name : chris} \\
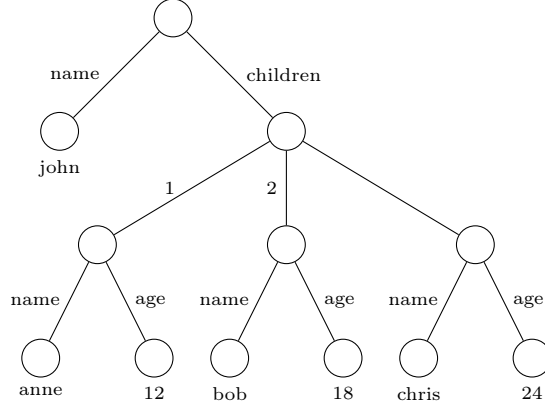&\text{children.3.age : 24}
\end{aligned}
$$

□

Figure 2: Object o from Example 2.1 as a tree.

Formally, we define an *object description* to be any set of pairs of the form $p : v$, where p is a path and v is an atomic value. If o is an object, the object description of o, denoted by $\mathrm{OD}(o)$, is defined inductively as follows:

- If o is $\emptyset$, or a singleton object of the form $\{k : b\}$ with b an atomic value, then $\mathrm{OD}(o) = o$.

- If o is a singleton object of the form $\{k : o'\}$, with o' an object, then

$$\mathrm{OD}(o) = \{k.p : b \mid (p : b) \in \mathrm{OD}(o')\}.$$

- If o is a non-singleton object, then

$$\mathrm{OD}(o) = \bigcup \{\mathrm{OD}(\{k : v\}) \mid (k : v) \in o\}.$$

Remark 2.5. Not every finite object description is the object description of some object; those that are, are called *proper*. Simple examples of improper object descriptions are $\{a : 1, a : 2\}$ and $\{a : 1, a.a : 1\}$. We will focus on proper object descriptions in Section 5. For now, we allow arbitrary object descriptions.

Vocabularies, instances, and queries We can finally define the fundamental notions of database instance and query in our data model. Just like a relational database instance is a finite collection of named relations, here we will define an instance as a finite collection of named object descriptions. Since object descriptions are binary relations (sets of pairs), we refer to their names as "relation names".

Formally, a *vocabulary* $\mathcal{V}$ is a finite set of relation names. An *instance* I over $\mathcal{V}$ assigns to each name $R \in \mathcal{V}$ an object description $I(R)$. Given two disjoint vocabularies $\mathcal{V}_{\mathrm{in}}$ and $\mathcal{V}_{\mathrm{out}}$, a *query from* $\mathcal{V}_{\mathrm{in}}$ *to* $\mathcal{V}_{\mathrm{out}}$ is a partial function from instances over $\mathcal{V}_{\mathrm{in}}$ to instances over $\mathcal{V}_{\mathrm{out}}$.

In database theory one often focuses on *generic* queries [2]. We can define a similar notion of genericity here. Let f be a permutation of **dom**. Then f can be extended to paths, packed keys, object descriptions, and instances, simply by applying f to every occurrence of an atomic key. Let

6

C be a finite subset of **dom** (these are the atomic keys that would be explicitly mentioned in a program for the query). Then a query Q is called C-*generic* if for every permutation f of **dom** that is the identity on C, and for every instance I, we have $Q(f(I)) = f(Q(I))$. In particular, if $Q(I)$ is undefined, then $Q(f(I))$ must also be undefined.

3   J-Logic

In the syntax of J-Logic, we assume disjoint supplies of *atomic variables* (ranging over atomic keys) and *path variables* (ranging over paths). The set of all variables is also disjoint from **dom**. We indicate atomic variables as $@x$ and path variables as $\$x$.

Key expressions and *path expressions* are defined just like keys and paths, but with variables added in. Formally, we define the sets of key expressions and path expressions to be the smallest sets such that

- every atomic key is a key expression, called a *constant*;

- every atomic variable is a key expression; constants and atomic variables are also called *atomic key expressions*;

- if e is a path expression then $\langle e \rangle$ is a key expression, called a *packed key expression*;

- every nonempty finite sequence of key expressions and path variables is a path expression.

Recall that an atomic value is an atomic key or $\emptyset$. Now an *atomic term* is an atomic value or an atomic variable.

A *predicate* is an expression of the form $P(e : t)$, with P a relation name, e a path expression, and t an atomic term.

An *equality* is an expression of the form $e_1 = e_2$, with e_1 and e_2 path expressions.

Many of the following definitions adapt the standard definition of Datalog [2] to our data model.

An *atom* is a predicate or an equality. A *negated atom* is an expression of the form $\neg A$ with A an atom. A *literal* is an atom (also called a positive literal) or a negated atom (a negative literal).

A *body* is a finite set of literals.

A *rule* is an expression of the form $H \leftarrow B$, where H is a predicate, called the *head* of the rule, and B is a body. We define the *limited variables* of the rule as the smallest set such that

- every variable occurring in a positive predicate in B is limited; and

- if all variables occurring in one of the sides of a positive equality in B are limited, then all variables occurring in the other side are also limited.

A rule is called *safe* if all variables occurring in the rule are limited.

Finally, a *program* is a finite set of safe rules with stratified negation. We omit the definition of stratified negation, which is well known [2]. For our purposes in this paper, stratified negation suffices. A program is called *positive* if it does not use negation. We also assume familarity with the distinction between recursive and nonrecursive programs.

Semantics We have defined the notion of instance as an assignment of object descriptions to relation names. A convenient equivalent view of instances is as sets of *facts*. A fact is an expression of the form $R(p : v)$ with R a relation name, p a path, and v an atomic value. An instance I over vocabulary $\mathcal{V}$ is viewed as the set of facts

$$I = \{R(p : v) \mid R \in \mathcal{V} \text{ and } (p : v) \in I(R)\}.$$

A *valuation* is a function ν defined on a finite set of variables, that maps atomic variables to atomic keys and path variables to paths. We say that ν is *appropriate* for a syntactical construct (such as a path expression, a literal, or a rule) if ν is defined on all variables occurring in the construct. We can apply an appropriate valuation ν to a key or path expression e in the obvious manner: we substitute each variable by its image under ν and obtain a key or a path $\nu(e)$. Likewise, we can apply an appropriate valuation to a predicate and obtain a fact.

Let L be a literal, ν be a valuation appropriate for L, and I be an instance. The definition of when I, ν *satisfies* L is as expected: if L is a predicate, then the fact $\nu(L)$ must be in I; if L is an equality $e_1 = e_2$, then $\nu(e_1)$ and $\nu(e_2)$ must be the same path. If L is a negated atom $\neg A$, then I, ν must not satisfy A.

A body B is satisfied by I, ν if all its literals are. Now a rule $\mathbf{r} = H \leftarrow B$ is satisfied in I if for every valuation ν appropriate for $\mathbf{r}$ such that I, ν satisfies B, also I, ν satisfies H.

The notions of EDB and IDB relation names of a program are well known: the IDB relation names are the relation names used in the head of some rules; the other relation names are the EDB relation names. Given a vocabulary $\mathcal{V}_{\text{in}}$, a program is said to be *over* $\mathcal{V}_{\text{in}}$ if all its EDB relation names belong to $\mathcal{V}_{\text{in}}$, and its IDB relation names do not.

Now the semantics of programs with stratified negation is defined as usual [2]. Recall that a program is called semipositive if negative predicates only use EDB relation names. We first apply the first stratum, which is semipositive, and then apply each subsequent stratum as a semipositive program to the result of the previous stratum. So we only need to give semantics for semipositive programs.

Let $\mathbf{P}$ be a semipositive program over $\mathcal{V}_{\text{in}}$, and let I be an instance over $\mathcal{V}_{\text{in}}$. Let $\mathcal{V}$ be the set of IDB relation names of $\mathbf{P}$. Then $\mathbf{P}(I)$ is the smallest instance over $\mathcal{V}_{\text{in}} \cup \mathcal{V}$ that satisfies all the rules of $\mathbf{P}$, and that agrees with I on $\mathcal{V}_{\text{in}}$.

In the end, a program $\mathbf{P}$ over $\mathcal{V}_{\text{in}}$ can be used to compute a query Q from $\mathcal{V}_{\text{in}}$ to $\mathcal{V}_{\text{out}}$, for any designated subset $\mathcal{V}_{\text{out}}$ of the IDB relation names of $\mathbf{P}$. Here, $Q(I)$ simply equals the restriction of $\mathbf{P}(I)$ to $\mathcal{V}_{\text{out}}$.

Syntactic sugar We have kept the syntax of J-Logic minimal so as to keep the formal definitions as simple as possible. For writing practical programs, however, it is convenient to introduce some syntactic sugar:

- Variables of the form $\%u$ range over atomic values, i.e., atomic keys or $\emptyset$. We could always eliminate such a variable in a rule by splitting the rule in two: one in which we replace $\%u$ by a normal atomic variable $@u$, and one in which we replace $\%u$ by $\emptyset$ (and resolve equalities accordingly).

- Variables of the form $?z$ range over paths or the empty sequence (recall that paths are nonempty). As long as such a variable is only used concatenated with other path expressions, we could always eliminate it from a rule by splitting the rule in two: one in which we

8

replace $?z$ by a normal path variable $\$z$, and one in which we simply delete all occurrences of $?z$ (resolving equalities accordingly).

- Variables of the form $\#z$ range over keys, atomic as well as packed. We could always eliminate such a variable in a rule by splitting the rule in one where we replace $\#z$ by an atomic variable $@z$, and one where we replace $\#z$ by a packed key expression $\langle \$z \rangle$.

Examples We aim to illustrate that J-Logic does not need recursion to express many useful queries involving deeply nested data. We begin, however, by illustrating why nonrecursive programs are desirable.

Example 3.1 (Nontermination). Due to the use of concatenation in heads of rules, the result of a recursive program applied to a finite instance may be infinite. A simple example is the following: (this program has no EDB relation names; the body of the first rule is empty)

$S(a : \emptyset) \leftarrow$
$S(a.\$x : \emptyset) \leftarrow S(\$x : \emptyset)$

We consider such programs to be nonterminating. For limited forms of recursion that guarantee termination or even tractability, we refer to the work of Bonner and Mecca [9, 29]. Nonrecursive programs clearly always terminate. $\qquad\square$

Example 3.2 (Deep equality). The following nonrecursive program is applied to the object description R of an object o, assumed to have values $o.a$ and $o.b$. The program tests equality of $o.a$ and $o.b$; if so, it outputs the fact $Q(\text{yes} : \emptyset)$ and if not, it outputs no facts. Note that $o.a$ and $o.b$ may be atomic keys (case handled by the first and last rule), or may be objects themselves. Thus, the other rules of the program test set equality of the object descriptions of $o.a$ and $o.b$.

$T(\text{atomic} : \emptyset) \leftarrow R(a : \%u), R(b : \%v)$
$Q'(\text{no} : \emptyset) \leftarrow R(a.\$x : \%u), \neg R(b.\$x : \%u)$
$Q'(\text{no} : \emptyset) \leftarrow R(b.\$x : \%u), \neg R(a.\$x : \%u)$
$Q(\text{yes} : \emptyset) \leftarrow \neg T(\text{atomic} : \emptyset), \neg Q'(\text{no} : \emptyset)$
$Q(\text{yes} : \emptyset) \leftarrow R(a : \%u), R(b : \%u)$ $\qquad\square$

Example 3.3 (Unnesting). Let o be the object described by R. The following single-rule program retrieves all subobjects of o (at arbitrary depths, but not o itself) that have a 'name'-value equal to 'John'. These objects are returned as top-level elements of a result object S, with new keys generated by packing.

$S(\langle \$x \rangle.\$y : \%u) \leftarrow R(\$x.\text{name} : \text{John}), R(\$x.\$y : \%u)$

$\qquad\square$

Example 3.4 (Key lookup, nesting). Like the previous example, the following program again considers subobjects, but now focuses on those that have a key 'ref' with an atomic key as value. That key is looked up and all values found for it are collected in a new subobject created under the 'ref' key. As in the previous example, new keys (for the elements of the collection) are generated using packing. The output object S is thus an "enrichment" of the input object R.

$T(\$x.\text{ref} : \emptyset) \leftarrow R(\$x.\text{ref} : @k)$
$S(\$x.\text{ref}.\langle \$y \rangle.?z : \%u) \leftarrow R(\$x.\text{ref} : @k), R(\$y.@k.?z : \%u)$
$S(\$x' : \%u) \leftarrow R(\$x' : \%u), \neg T(\$x' : \emptyset)$

9

4 Expressiveness and complexity

Nonrecursive J-Logic has polynomial-time data complexity. Since rules are safe, we can find valuations satisfying the body of a rule through finding valuations of predicates. To find the valuations satisfying a predicate $P(e : t)$, note that the path expression e is a sequence of key expressions and path variables. Let k be the length of this sequence. Then we choose a pair $(p : v)$ from P; the number of possibilities is linear. We match t to the atomic value v in the obvious manner, and match e to the path p by splitting p in k pieces. The number of possible splits is polynomial of degree k. A piece corresponding to a path variable provides a binding for that path variable, or must be equal to an already existing binding. A piece corresponding to an atomic variable must be an atomic key. A piece corresponding to a constant must match the constant. Finally, a piece corresponding to a packed key expression $\langle e' \rangle$ must be a packed key $\langle p' \rangle$. Then e' is matched to p' in turn.

Using positive, recursive, programs we can already simulate Turing machines [9]. Using general programs, we are computationally complete: we can express any computable C-generic query from finite instances to finite instances. Using an encoding of instances as defined here by relational database instances, this can be proven following the known body of work on the computational completeness of query languages [14, 4, 3, 36, 13].

Nonrecursive programs, relational algebra, and practical languages Let us call a class of instances *bounded* if there is a fixed bound on the length of all paths occurring in the instances, as well as on the nesting depth of packed keys.

On a bounded class of inputs, nonrecursive J-Logic can be simulated by relational algebra. Indeed, due to the bound, there are only finitely many nonequivalent predicates, and each equivalence class can be described using atomic variables only. Thus, for each equivalence class of predicates we can keep the bindings in a fixed-arity relation. Given such a representation the evaluation of a rule can be expressed in relational algebra. Moreover, the application of a rule to a bounded instance produces again a bounded instance (with the new bound depending only on the old bound and the rule). In this way we can simulate nonrecursive J-Logic over bounded instances in relational algebra.

Conversely, it is quite clear that we can represent all relational database instances over some fixed schema as a bounded class of instances in our data model. There are various ways to do this. One approach is to represent a tuple as an object in the obvious way (each attribute is a key) and then represent a set of tuples as a set of objects, using tuple identifiers as top-level keys. Under such a representation we can easily simulate, say, the relational algebra, using nonrecursive J-Logic. We use packing to generate new tuple identifiers, as illustrated in the Introduction for Cartesian product.

Another approach is to use a (bounded-depth) trie representation for relations, as used, for example, in the Leapfrog Triejoin algorithm [37]. Such tries are naturally represented as JSON objects. We can then again simulate the relational algebra using nonrecursive J-Logic, and we would not even need packing.

Note that practical JSON query languages SQL++ [30] and JSONiq without recursive functions [18] are mainly geared towards bounded-depth data. Apart from features such as aggregation and full-text search, these languages are fundamentally based on the nested relational algebra or calculus [12]. This calculus can be translated into nonrecursive J-Logic. As already mentioned in the Introduction, packing can be used to represent nested collections. The only caveat (which is also

not really mentioned by SQL++ and JSONiq) is to do duplicate elimination on nested collections. It follows from known results [35] that a special set-oriented packing operator would need to be added for this purpose.

Moreover, we feel that the main contribution of J-Logic is as a language in which nonrecursive programs can also work well with unbounded inputs, i.e., deeply nested data.

Limitations of nonrecursive programs   The above discussion immediately yields examples of queries not expressible by nonrecursive programs: any query over relational instances that is not expressible in the relational algebra will do, such as the transitive closure of a binary relation. That does not tell us anything about unbounded instances, however. In Proposition 4.2 we will give a general necessary condition on the output of nonrecursive programs.

Example 4.1. Let c be some constant and consider the query Q from $\{R\}$ to $\{S\}$ defined by

$$Q(I) = \{S(k_1.c.k_2.c \ldots k_n.c : \emptyset) \mid R(k_1.k_2 \ldots k_n : \emptyset) \in I\}$$

where n is not fixed but ranges over all possible lengths. Proposion 4.2 will imply that this query is not expressible by a nonrecursive program.   □

Bonner and Mecca [9] have proposed mixing transducers with sequence Datalog, so that manipulations as in the above example can be easily expressed. They already noted informally that without recursion through concatenation, only a fixed number of concatenations can be performed. The following proposition formalizes this observation and adapts it to J-Logic.

In order to state the necessary condition, we introduce the following notations. For a set S of paths, $sub(S)$ denotes all subpaths of paths occurring in S (also paths occurring in packed keys). Also, $concat(S, i)$ denotes all paths that can be built up (using concatenation and packing) from the paths in S using a total of at most i concatenations. The set of paths of an instance J is denoted by $paths(J)$, so formally, $paths(J) = \{p \mid R(p : v) \in J$ for some R and $v\}$.

Proposition 4.2. *Let* $\mathbf{P}$ *be a nonrecursive program. There exists a finite set* L *of paths and a natural number* i *such that for every instance* I, *we have* $paths(\mathbf{P}(I)) \subseteq concat(sub(paths(I) \cup L), i)$.

Proof. By induction on the number of strata. For the base case, assume $\mathbf{P}$ consists of a single stratum. By an obvious rewriting we may assume without loss of generality that the body of each rule only mentions EDB relation names. Consider an element $p \in paths(\mathbf{P}(I))$. Then p is produced by applying a valuation to a path expression, say e, in the head of some rule. Every variable is mapped to an element of $sub(paths(I))$. Let $\hat{e}$ denote the sequence obtained by removing all variables from e, as well as all opening and closing brackets of packed keys; we refer to these lexical elements as *separators*. Let i_e denote the number of separators; we can view e as chopping $\hat{e}$ in $i_e + 1$ pieces. Thus, $p \in concat(sub(paths(I) \cup \{\hat{e}\}), i_e + 1)$. Hence, we can set i to the maximum i_e, and we can set L to the set of $\hat{e}$.

Now assume $\mathbf{P}$ has at least two strata. Let $\mathbf{P}'$ be the part without the last stratum, which we denote by $\mathbf{P}''$. So, $\mathbf{P}$ is the composition of $\mathbf{P}''$ after $\mathbf{P}'$. By induction, we have i' and L' for $\mathbf{P}'$. Moreover, reasoning as in the base case, we have i'' and L'' for $\mathbf{P}''$ applied to $\mathbf{P}'(I)$. After some calculations we can see that we can now set $i = i' \cdot i''$ and $L = L' \cup L''$.   □

Flat–flat queries   An instance is called *flat* if no packed keys occur in it. A query Q is called *flat–flat* if for every flat instance I, if $Q(I)$ is defined then it is also flat. It may still be convenient to use packing in the computation of a flat–flat query, as illustrated next.

Example 4.3. The query from Example 4.1 is flat–flat. Over flat inputs, we can compute it by the following program:

$$T(\langle @i \rangle.?y : \emptyset) \leftarrow R(@i.?y : \emptyset)$$
$$T(?x.@i.c.\langle @j \rangle.?y : \emptyset) \leftarrow T(?x.\langle @i \rangle.@j.?y : \emptyset)$$
$$S(?x.@i.c : \emptyset) \leftarrow T(?x.\langle @i \rangle : \emptyset)$$

We see that packing is conveniently used as a cursor to run through the sequence. With more effort, however, we can also compute the query without using packing. The trick is to use some constant a and to look for the longest consecutive sequence of a's occurring in any path in R. Then a sequence of a's one longer than that can be used as a cursor. The program is as follows. Since all predicates in the program will be of the form $P(e : \emptyset)$, we abbreviate them as $P(e)$.

$$Sub(a.?y) \leftarrow R(?x.a.?y.?z)$$
$$Subnota(\$x.@i.?y) \leftarrow Sub(\$x.@i.?y), \ @i \neq a$$
$$Suba(\$x) \leftarrow Sub(\$x), \neg Subnota(\$x)$$
$$A(a.\$x) \leftarrow Suba(\$x), \neg Suba(a.\$x)$$
$$T(\$a.\$x) \leftarrow R(\$x), A(\$a)$$
$$T(?x.@i.c.\$a.?y) \leftarrow T(?x.\$a.@i.?y), A(\$a)$$
$$S(\$x) \leftarrow T(\$x.\$a), A(\$a) \qquad \qquad \square$$

The above example illustrates a general theorem:

Theorem 4.4 (Flat–flat theorem). *For every J-Logic program computing a flat–flat query there is equivalent program without packing, over flat instances.*

Proof. The proof is easy if we can use two constants, say a and b, that are never used in any instance. Then a packed key expression $\langle e \rangle$ can be simulated using $a.e.b$, where we also would need to write additional rules checking that e matches a path with balanced a's and b's.

If we want a simulation that always works, without an assumption on the constants used in instances, we can encode a path $p = k_1.k_2 \ldots k_n$ by its doubled version $p' = k_1.k_1.k_2.k_2 \ldots k_n.k_n$. Then $\langle e \rangle$ can be simulated using $a.b.e'.b.a$. For example, the path $a.c.\langle a.b \rangle.b.a$ is encoded as $a.a.c.c.a.b.a.a.b.b.b.a.b.b.a.a$. This encoding can be computed without packing using the technique illustrated in Example 4.3. Assume we want to encode the contents of a relation A and have in relation A_c computed with this technique a path of c's that is one longer then the longest path in relation A. Let us call this path c^{+1}. We can then define a program, that computes the encoding of A. This program starts with copying A but adds $d.c^{+1}.c^{+1}.d$ as a cursor with d different from c, and then moves this cursor to the left while doubling constants. Note that we cannot use $d.c^{+1}.d$ as a cursor since we are doubling paths and so might be creating subpaths equal to $d.c^{+1}.d$. We can also not simply use $c^{+1}.c^{+1}$ since the original path might contain c's and so there might be uncertainty while matching about where the cursor begins and ends.

$$A_1(\$x.d.c^{+1}.c^{+1}.d : \%u) \leftarrow A(\$x : \%u)$$
$$A_1(?x.d.c^{+1}.c^{+1}.d.@i.@i.?y : \%u) \leftarrow A_1(?x.@i.d.c^{+1}.c^{+1}.d.?y : \%u)$$

Recall that we are encoding the input of a flat-flat query, and so can assume the input contains no packing that needs to be encoded. As a final step we then select those paths where the cursor has arrived at the beginning and remove the cursor, which produces the encoding of A in A_2.

$$A_2(\$x : \%u) \leftarrow A_1(d.c^{+1}.c^{+1}.d.\$x : \%u)$$

We can transform the original program with packing to one that does not use packing and assumes that the input is encoded as previously described. This transformation is done as follows:

- Any constant and atomic variables in a path expression are doubled, like in the encoding. So a constant a is replaced with $a.a$, and an atomic variable $@i$ is replaced with $@i.@i$. Note that in a predicate $P(e : t)$ we do not replace constants and atomic variables in t.

- Any path variable is left in place, but the clause is extended with a check $Enc_B(\$x)$ with B a relation name of a predicate in which $\$x$ occurs in the rule, to see if the variable $\$x$ matches a subpath of a path in B that is a valid encoding.

- Any packed key expression $\langle e \rangle$ is replaced with $a.b.e'.b.e$ where e' is the transformation of e.

As an example, consider the following rule:

$$A(@v.a.\$y : @w) \leftarrow B(\$x.\langle c.\$y \rangle.@w : @v), \neg C(@v.b : \emptyset)$$

It is translated to:

$$A(@v.@v.a.a.\$y : @w) \leftarrow B(\$x.a.b.c.\$y.b.a.@w.@w : @v), \neg C(@v.@v.b.b : \emptyset), Enc_B(\$x), Enc_B(\$y)$$

The predicate Enc_B can be expressed by a program without packing as follows:

$$Enc_B(a.b.?x.b.a) \leftarrow B(?u.a.b.?x.b.a.?v), Enc_B(?x)$$
$$Enc_B(@i.@i.?x) \leftarrow B(?u.@i.@i.?x.?v), Enc_B(?x)$$
$$Enc_B(?x.@i.@i) \leftarrow B(?u.?x.@i.@i.?v), Enc_B(?x)$$

It is clear that the transformed program simulates the original program on encoded instances.

As the following step we need to show that the encoded result can be decoded without using packing. So let B be a relation in $\mathcal{V}_{\text{out}}$ that contains an encoded result, and assume that with the technique of Example 4.3 we have computed in B_c a path of c's that is one longer then the longest path of c's in B. We will denote this path of c's as c^{+1}. The approach is basically the same as for the encoding: we place a cursor in each path to indicate until how far we have decoded the path. The first program copies B but adds $d.c^{+1}.d$ as a cursor, and then moves this cursor to the left while undoubling constants.

$$B_1(\$x.d.c^{+1}.d : \%u) \leftarrow B(\$x : \%u)$$
$$B_1(?x.d.c^{+1}.d.@i.?y : \%u) \leftarrow B_1(?x.@i.@i.d.c^{+1}.d.?y : \%u)$$

Recall that we are decoding the output of a flat-flat query, and so can assume the input contains no encoded packing that needs to be decoded. As a final step we select the paths where the cursor has arrived at the beginning and remove the cursor, which produces the decoding of B in B_2.

$$B_2(\$x : \%u) \leftarrow B_1(d.c^{+1}.d.\$x : \%u)$$

$\square$

The above proof needs recursion, even if the given program is nonrecursive. In general it is fair to say that the above flat–flat theorem is mainly of theoretical interest. Still it is an interesting open question whether for every nonrecursive program computing a flat–flat query, there is an equivalent *nonrecursive* program without packing, over all flat instances.

5   Proper object descriptions and object–object queries

In Remark 2.5 we introduced the notion of proper object description as the object description of an actual object, as opposed to just any set of path–value pairs. Proper object descriptions can be characterized as follows.

Proposition 5.1. *A finite object description D is proper if and only if it satisfies the following two constraints:*

- *the functional dependency from paths to atomic values, i.e., if $(p : u) \in D$ and $(p : v) \in D$, then $u = v$.*

- *prefix-freeness, i.e., if p and q are paths, and $(p.q : u) \in D$ for some u, then $(p : v) \notin D$ for every v.*

Proof. The only-if direction is clear. The if-direction can be proven by induction on the maximum length of a path in D. If this maximum equals 1, then D clearly describes an object with only atomic values. The object is well-defined thanks to the functional dependency. Now assume the maximum is at least 2. We construct an object o such that $\text{OD}(D) = o$ as follows.

Define K_1 as the set of keys k such that $(k : v) \in D$ for some v. Thanks to the functional dependency v is unique for k and we denote v by $D(k)$. As in the base case, we obtain an object o_1 defined on K_1 defined by $o_1.k = D(k)$.

Define K_2 as the set of atomic keys k such that $(k.p : v) \in D$ for some path p and atomic value v. For each $k \in K_2$ define the object description $D_k = \{(p : v) \mid (k.p : v) \in D\}$. Then D_k has a shorter maximum path length and still satisfies the two constraints. Hence, by induction, D_k describes an object o_k. We now define the object o_2, defined on K_2, by setting $o_2.k = o_k$.

Thanks to prefix-freeness, K_1 and K_2 are disjoint. Hence the union $o_1 \cup o_2$ is a well-defined object and yields the desired object o. □

Example 5.2. $D = \{a : 1, a.a : 1\}$ is not prefix-free and indeed D is not proper. In proof, suppose D would be the description of an object o. Then $o.a$ is the atomic value 1 by the first pair in D. But by the second pair, $o.a$ is an object with a-value 1, a contradiction. □

Remark 5.3. Only finite object descriptions can be proper, since objects are always finite. Still, the two constraints from the above proposition can be taken to be the *definition* of properness for infinite instances. Later in this paper, we will consider the object–object problem, the implication problem for jaegds, and the containment problem. These three problems ask a question about *all* instances. These problems do not change, however, if we restrict attention to finite instances. □

An instance is called proper if it assigns a proper object description to every relation name. A query Q is called *object–object* if for every proper instance I, if $Q(I)$ is defined then it is also proper.

The object–object property is practically important. In practice, a JSON processor may accept improper object descriptions, or object syntax that is not well-defined, such as $\{a : 1, a : 2\}$ or $\{a : 1, a : \{b : 2\}\}$. However, the processor will interpret such syntax in an unpredictable manner. Perhaps it will overwrite a previously read a-value by an a-value read later. Or, on the contrary, it may keep only the value that was read first. To avoid depending on such system-defined behavior, we better write queries having the object–object property.

One may go further and demand that also all intermediate relations generated by a J-Logic program hold proper object descriptions. This may be relevant, for example, if we implement the query language on top of a JSON store. We next show that this is always possible. We call the

result the "object–object theorem", which may be a bit pompous, as it is proven by a simple trick using packing (thus again illustrating the utility of packing).

Theorem 5.4 (Object–object theorem). *Let* **P** *be a program expressing an object–object query* Q. *Then there exists an equivalent program* **P**′ *such that, on any proper input instance, all IDB relations of* **P**′ *hold proper object descriptions. Program* **P**′ *has the same number of strata as* **P**, *and is recursive only if* **P** *is.*

Proof. The idea is to encode arbitrary object descriptions by object descriptions that are always proper. Then the program is simulated using the encoding. At the end the output relations are decoded. Such an encoding is easy to do using packing.

Formally, fix an arbitrary atomic key b. For any input relation name R we introduce the following two encoding rules:

$R'(\langle \$x \rangle . \langle @u \rangle : \emptyset) \leftarrow R(\$x : @u)$
$R'(\langle \$x \rangle . \langle b.b \rangle : \emptyset) \leftarrow R(\$x : \emptyset)$

These rules are added to the first stratum of **P**.

Furthermore, we modify **P** by replacing each atom (in bodies and in heads) of the form $P(e : t)$ by $P'(\langle e \rangle . \langle t \rangle : \emptyset)$ if t is not $\emptyset$, and by $P'(\langle e \rangle . \langle b.b \rangle : \emptyset)$ otherwise.

Finally for every output relation name S we add the following decoding rules to the last stratum:

$S(\$x : @u) \leftarrow S'(\langle \$x \rangle . \langle @u \rangle)$
$S(\$x : \emptyset) \leftarrow S'(\langle \$x \rangle . \langle b.b \rangle)$ □

Example 5.5. The following program begins by eliminating the top layer from an object R, which brings the second-level keys to the top level. This intermediate result R_1 may well be improper. We then throw away all "bad" paths (paths that violate properness). The result, S, is of course proper. Thus, this program computes an object–object query but is easiest to write using improper intermediate results. Yet, the object–object theorem assures us it can be rewritten using only proper intermediate results.

$R_1(\$y : \%u) \leftarrow R(\#x.\$y : \%u)$
$Bad(\$y : \%u) \leftarrow R_1(\$y : \%u), R_1(\$y : \%v), \%u \neq \%v$
$Bad(\$x.\$z : \%v) \leftarrow R_1(\$x : \%u), R_1(\$x.\$z : \%v)$
$S(\$y : \%u) \leftarrow R_1(\$y : \%u), \neg Bad(\$y : \%u)$ □

Remark 5.6. Our proof of the object–object theorem uses packing. Of course there is nothing wrong with packing; we think it is a versatile tool. Yet, theoretically one may wonder whether one can also do without. Indeed it turns out one can prove a combination of the flat–flat theorem and the object–object theorem. Specifically, for every program computing a flat–flat object–object query, we can find a program without packing that is equivalent over flat instances and that only works with proper intermediate results. The idea is to encode a path–value pair $a_1 \ldots a_n : \emptyset$ by $b.a_1 \ldots b.a_n.a.a.b : \emptyset$, and a path–value pair $a_1 \ldots a_n : c$ by $b.a_1 \ldots b.a_n.c.a.a.b : \emptyset$. It can be verified that an encoding of an object description is always proper. The program is then modified to work over encodings. As for the flat–flat theorem, the program without packing would need recursion. Again we leave open whether there is a nonrecursive version of the flat–flat object–object theorem.

5.1 Deciding the object–object property

The *object–object problem* is to decide, given a J-Logic program $\mathbf{P}$ and appropriate vocabularies $\mathcal{V}_{\text{in}}$ and $\mathcal{V}_{\text{out}}$, whether the query from $\mathcal{V}_{\text{in}}$ to $\mathcal{V}_{\text{out}}$ computed by $\mathbf{P}$ has the object–object property.

In general, this problem is of course undecidable. It is undecidable for positive recursive programs, because these can simulate Turing machines, and also for nonrecursive programs that can use negation, because these can express first-order logic (relational algebra).

Another restriction we will introduce concerns the use of equations in programs. These can sometimes add expressive power that is usually associated with recursive programs.

Example 5.7. The following program selects from R all paths that contain only a's.

$$S(\$x : \%u) \leftarrow R(\$x : \%u), a.\$x = \$x.a \quad \square$$

To rule out such programs we introduced the following definitions. Given a rule $H \leftarrow B$ we define the *equation graph* as an undirected multigraph where all variables in B are the nodes and the number of edges between variable x and variable y is equal to the sum of $\#_x(e_1) \times \#_y(e_2)$ for each distinct equation $e_1 = e_2$ in B, where $\#_x(e)$ denotes the number of times variable x occurs in e. We call a nonempty sequence of edges in an equation graph a *path* if in the sequence each two subsequent edges are incident. We call a path a *cycle* if the first and last edge are incident, and an equation graph *cyclic* if it contains a cycle. We say that the rule $H \leftarrow B$ is *equationally cyclic* if the equation graph associated with B contains a cycle. We call a program *equationally cyclic* if at least one of its rules is cyclic, and *equationally acyclic* if there is no such rule.

Example 5.8. The following rule is equationally cyclic:

$$S(\$x : \%u) \leftarrow R(\$x : \%u), a.\$x = \$y, \$y = \$x.a$$

This is because its equation graph contains the cycle $\langle \{\$x, \$y\}_{a.\underline{\$x=\$y}}, \{\$y, \$x\}_{\underline{\$y=a.\$x}} \rangle$. The subscript of each edge indicates which equation and occurrences the edge corresponds to.

Also the following rule has a cyclic equation graph:

$$S(\$x : \%u) \leftarrow R(\$x : \%u), \$y.\$y = \$x$$

This is because it contains the cycle $\langle \{\$x, \$y\}_{\underline{\$y.\$y=\$x}}, \{\$y, \$x\}_{\$y.\underline{\$y=\$x}} \rangle$.

The program in Example 5.7 is also equationally cyclic since the equation graph of its rule contains the cycle $\langle \{\$x\}_{a.\underline{\$x=\$x}.a} \rangle$. $\quad\blacksquare$

This restriction on program allows us to formulate the main result of this subsection:

Theorem 5.9. *The object–object problem is decidable for positive, nonrecursive programs where all rules are equationally acyclic and their head contains every variable at most once.*

Our starting point is to note that this problem has similarities with a problem known from relational databases. This problem is the FD–FD implication problem for (unions of) conjunctive queries (UCQs) [1, 2]. It is also called the view dependency problem [27]. This problem asks, given two sets Σ_1 and Σ_2 of functional dependencies (FDs) and a query Q, whether the result of Q, applied to an instance satisfying Σ_1, always satisfies Σ_2. The similarity lies in that properness involves satisfying an FD; moreover, positive nonrecursive J-Logic programs that are equationally acyclic are the J-Logic analog of UCQs. Of course there are also differences: J-Logic has packing and path variables, and the notion of properness is not only about FDs but also about prefix-freeness.

The decidability of the FD–FD implication problem for UCQs follows readily from the decidability of the implication problem for equality-generating dependencies (egds), using the chase [2, 8]. Hence our approach is to introduce *J-Logic atomic equality-generating dependencies* or jaegds, and investigate the chase for these dependencies.

Syntactically, a *jaegd* is a rule σ of the form $B \to E$, where B is a positive body without equalities and E is an atomic equality, i.e., an equality of the form $u = v$ where u and v are atomic key expressions (atomic constants or atomic variables). If u or v is a variable, that variable must occur in B.

Semantically, note that B consists exclusively of positive predicates. Hence, for any instance I and valuation ν appropriate for B, we have that I, ν satisfies B if and only if $\nu(B) \subseteq I$. We denote this by $\nu : B \to I$ and call ν a *matching* of B in I. We now define that I satisfies a jaegd σ as above, denoted by $I \models \sigma$, if for every matching $\nu : B \to I$, the atomic keys $\nu(u)$ and $\nu(v)$ are identical.

Note that dependencies of the form $B \to a = b$, where a and b are distinct atomic keys, are allowed. Since $a = b$ is always false, this can be written more clearly as $B \to \textbf{false}$ or also $B \to \bot$. This is used to express a *denial constraint*: it is only satisfied in an instance I if there does not exist any matching of B in I.

Note that we also allow dependencies of the form $B \to u = u$. Obviously such dependencies are trivial (satisfied in any instance), but we allow them because they may be produced by the chase procedure.

Example 5.10. By Proposition 5.1, an object description D is proper if and only if it satisfies the jaegds δ_1–δ_6:

$$\delta_1 : D(\$x : @i), D(\$x : @j) \to @i = @j$$
$$\delta_2 : D(\$x : \emptyset), D(\$x : @i) \to \bot$$
$$\delta_3 : D(\$x : @i), D(\$x.\$y : \emptyset) \to \bot$$
$$\delta_4 : D(\$x : @i), D(\$x.\$y : @j) \to \bot$$
$$\delta_5 : D(\$x : \emptyset), D(\$x.\$y : \emptyset) \to \bot$$
$$\delta_6 : D(\$x : \emptyset), D(\$x.\$y : @j) \to \bot$$

$\square$

For a set of dependencies Σ, we define $I \models \Sigma$ to mean that I satisfies every dependency in Σ. We say that Σ *logically implies* a dependency σ if every instance that satisfies Σ also satisfies σ. The *implication problem for jaegds* asks to decide, given a set of jaegds Σ and a jaegd σ, whether Σ logically implies σ. We actually do not know whether this problem is decidable in general. We will, however, solve a special case that is sufficient to solve the object–object problem.

The Chase   We first need the notion of a *variable mapping*. This is a function defined on a finite set of variables that maps path variables to path expressions and atomic variables to atomic key expressions. Like valuations, we can apply a variable mapping to a predicate simply by applying it to every variable occurring in the predicate. The result is again a predicate. Thus, the result of applying a variable mapping to a body is again a body. A *homomorphism* h from a body B_1 in a body B_2, denoted by $h : B_1 \to B_2$, is a variable mapping defined on at least all variables in B_1 such that $h(B_1) \subseteq B_2$.

With this notion of homomorphism in place, the notion of *chasing a jaegd σ with a set of jaegds* Σ is defined entirely similarly to the well-known chase for egds in the relational model [2].

The Chase   Let Σ be a set of jaegds and let σ be a single jaegd. Let B be the body of σ. By *applying a chase step* we mean the following:

1. Pick a dependency $C \rightarrow (u = v)$ in Σ.

2. Pick a homomorphism $h : C \rightarrow B$ such that $h(u)$ and $h(v)$ are not identical.

3. We consider the possibilities:

   - If $h(u)$ and $h(v)$ are different atomic keys, we say that the chase step has failed.
   - If one of $h(u)$ and $h(v)$ is an atomic key and the other is a variable, we substitute the atomic key for the variable everywhere in σ.
   - If both $h(u)$ and $h(v)$ are variables, we substitute $h(u)$ for $h(v)$ everywhere in σ.

If we can apply a sequence of chase steps, starting in σ, and applying each subsequent step to the result of the previous step, until we can make the chase step fail, we say that *chasing σ with Σ fails*. If, in contrast, we can apply a sequence of chase steps without failure until no chase step can be applied anymore, we say that chasing σ with Σ succeeds. An infinite sequence of chase steps is not possible, because we only equate atomic variables to atomic keys and the number of atomic variables and keys appearing in B is finite.

It is not difficult to see that the chase is locally confluent, whence confluent by Newman's Lemma. Hence, given the above definitions, it is not possible for the chase to succeed and fail at the same time.

The chase provides a sound proof procedure for logical implication, as stated in the following proposition.

Proposition 5.11. *Assume that either chasing σ with Σ fails, or the chase succeeds and results in a jaegd whose consequent is a trivial equality. Then Σ logically implies σ.*

The proof for this proposition is essentially the same as for egds in the relational model. It starts with the following property, which expresses soundness of the chase procedure.

Lemma 5.12. *If the chase fails, then σ is vacuously true under Σ, i.e., for every instance I satisfying Σ, there exists no matching of B in I. If the chase succeeds with a final result σ', then σ and σ' are equivalent under Σ, i.e., for every instance I satisfying Σ, we have $I \models \sigma$ if and only iff $I \models \sigma'$.*

The above lemma implies the following:

Proof of Proposition 5.11. Let I be an instance satisfying Σ. We must show $I \models \sigma$. Thereto consider a matching $\alpha : B \rightarrow I$. By Lemma 5.12, chasing σ by Σ succeeds (otherwise the matching α cannot exist). We are given that the chase yields a dependency σ' with a trivial equality as a consequent. Hence, trivially $I \models \sigma'$. However, by Lemma 5.12, this implies also $I \models \sigma$ as desired. □

For egds in the relational model, the converse to the above proposition holds as well, showing the completeness of the chase as a proof procedure. In our model, however, the converse fails, as shown next.

Example 5.13. Consider Σ consisting of the following three denial constraints:

$$P(@x : \emptyset) \rightarrow \bot$$
$$P(\langle \$x \rangle : \emptyset) \rightarrow \bot$$
$$P(\$x.\$y : \emptyset) \rightarrow \bot$$

18

Then Σ is equivalent to the single denial constraint $\sigma \equiv P(\$x : \emptyset) \to \bot$, so certainly Σ logically implies σ. However, chasing σ with Σ does not fail. Actually, no chase step can be applied at all and the chase ends immediately on σ itself. Since the consequent $\bot$ is not a trivial equality, this shows that the converse of Proposition 5.11 fails. $\qquad\square$

We can still get completeness of the chase in a special case, which we call *unambiguous*. We first define the notion of *weak variable mapping*. Recall that a variable mapping must map atomic variables to atomic key expressions. A weak variable mapping is like a variable mapping, except that atomic variables may also be mapped to path variables. A *weak morphism* from a body B_1 in a body B_2 is a weak variable mapping h such that $h(B_1) \subseteq B_2$.

Now consider an input (Σ, σ) to the implication problem for jaegds. We say that (Σ, σ) is *unambiguous* if either chasing σ with Σ fails, or the chase succeeds, and the following condition holds. Let B' be the body of the jaegd resulting from the chase. Then every weak morphism from a body in Σ to B' must actually be a variable mapping.

Example 5.14. Take Σ and σ from the previous example. We already noted that the chase succeeds immediately. We see there is a weak morphism from the body $\{P(@x : \emptyset)\}$ of the first dependency in Σ, to the body $\{P(\$x : \emptyset)\}$ of σ, namely the mapping $@x \mapsto \$x$. This is not a variable mapping. Hence (Σ, σ) is not unambiguous. $\qquad\square$

Example 5.15. For another example, consider the set $\Delta = \{\delta_1, \ldots, \delta_6\}$ from Example 5.10. Then (Δ, σ) is always unambiguous for any σ. Indeed, atomic variables occur only in the second component of predicates in Δ, i.e., after the : sign, and path variables can never occur after the : sign in any body. $\qquad\square$

The notion of unambiguity captures the cases where the usual proof of completeness of the chase applies in our setting. So, we have the following result.

Proposition 5.16. *Assume Σ logically implies σ, and (Σ, σ) is unambiguous. Then chasing σ with Σ fails, or the chase succeeds and results in a jaegd whose consequent is a trivial equality.*

Proof of Proposition 5.16. Let σ be of the form $B \to (w = z)$. Suppose the chase succeeds and results in the jaegd $\sigma' \equiv B' \to (w' = z')$. We must prove that w' and z' are identical.

We can view B' as an instance I by viewing each variable as an atomic key; it is customary to refer to these atomic keys as *frozen variables*. We claim that $I \models \Sigma$.

To prove the claim, consider a dependency $C \to (u = v)$ in Σ and a matching $\alpha : C \to I$. We can view α as a weak morphism from C to B'. Because (Σ, σ) is unambiguous, α does not map atomic variable to frozen path variables, i.e., it is really a homomorphism from C to B'. Since the chase succeeded with B' the body of the final result, there is no chase step possible in B'. This means that $\alpha(u)$ and $\alpha(v)$ must be identical and thus $I \models \phi$.

We now know that $I \models \Sigma$. Since we are given that Σ logically implies σ, also $I \models \sigma$. Recall that σ' is the result of subsequent applications of chase steps, starting from σ. Each chase step maps an atomic variable to another atomic variable or an atomic key, so amounts to applying a homomorphism. The composition of homomorphisms is also a homomorphism. Hence, there is a homomorphism from B to B' that maps w to w' and z to z'. This homomorphism can be viewed as a matching of B in I. Since $I \models \sigma$, the images of w and z must be identical. We conclude that w' and z' are identical as desired. $\qquad\square$

It follows that the unambiguous cases of the implication problem for jaegds are decidable by the chase. En route to solving the object–object problem, it is especially important that chasing from Δ is unambiguous, as we saw in Example 5.15.

19

Equality elimination   There is one final hurdle to overcome. A discrepancy between bodies of jaegds and bodies of positive J-Logic rules that are equationally acyclic is that the latter can have equalities. We next show, however, that equalities can always be removed.

Consider the equality $e_1 = e_2$ where $e_1 = \$x.@y.a.b.\$x$ and $e_2 = \$v.@w.b.\$u$. We define a notion of *unifier* as a variable mapping that, when applied as a substitution, maps two path expressions to the same path expression. For example, for e_1 and e_2 we have the following unifier: $u_1 = \{\$v \mapsto \$x.@y, @w \mapsto a, \$x \mapsto \$u\}$. Note that indeed $u_1(e_1) = \$x.@y.a.b.\$u = u_1(e_2)$. We say that a unifier u_1 is equal or more general than another unifier u_2 if there is a variable mapping u_3 such that $u_2(e) = u_3(u_1(e))$ for any path expression e. We call two unifiers equivalent if one is equal or more general than the other and vice versa. It is not hard to see that this defines a pre-order and moreover that if two unifiers are equivalent, they must be identical up to renaming the variables in the result. We will call a unifier a *most-general unifier* if all unifiers that are equal or more general are in fact equally general.

Then, we can observe the following:

Lemma 5.17. *Given an acyclic equality $e_1 = e_2$ where e_1 and e_2 then the set of most-general unifiers of e_1 and e_2 has finitely many equivalence classes.*

Proof. We start with considering a unifier of e_1 and e_2 that maps them both to a path expression e_3. For example, let us consider $e_1 = a.\$x.b.\$y.c$ and $e_2 = a.\langle\$u\rangle.\$v.\langle@w\rangle.b.c$. A possible unifier u that maps both to a path expression e_3 can be represented in a diagam as follows:

$$e_3 = a.\overbrace{\langle b.\underbrace{@d}\rangle}^{\$x}.a.\overbrace{\underbrace{\$e.b.\langle b.\$e\rangle.@d}}^{\$y}.\langle \overbrace{\underbrace{@d}}\rangle.b.c$$
$$\quad\quad\quad \underbrace{}_{\$u} \quad\quad \underbrace{}_{\$v} \quad\quad \underbrace{}_{@w}$$

We can observe that a fragment of p where two variables overlap, such as for example the fragment $\langle b.\$e\rangle.@d$ where $\$y$ and $\$v$ overlap, it holds that this fragment is well-balanced. This is because every opening bracket in the fragment must have a following matching closing bracket in the fragment, since the fragment of $\$v$ is well-balanced. Vice versa, every closing bracket in the fragment must have preceding matching opening bracket in the fragment, since the fragmetn of $\$y$ is well-balanced. Consequently, the fragment in the overlap is a path.

It follows that from the unifiers we can derive a more general unifier u' by replacing in the diagram every fragment where two variables overlap with a distinct fresh variable. This fresh is a path variable, unless one of the two overlapping variables is an atomic variable, in which case it is an atomic variable. In the previous example, this results in:

$$e_4 = a.\langle \overbrace{\underbrace{\$q}}^{\$x} \rangle.\overbrace{\underbrace{\$r}.b.\underbrace{\$s}}.\langle \overbrace{\underbrace{@t}}^{\$y} \rangle.b.c$$
$$\quad\quad\quad \underbrace{}_{\$u} \quad \underbrace{}_{\$v} \quad \underbrace{}_{@w}$$

It can be shown that the resulting diagram defines a unifier if $e_1 = e_2$ is acyclic. After all, if for two overlapping variables we make a replacement, it follows from acyclicity that each of these variables occurs at most once in e_1 and e_2. So there is only one place in p' that describes what these variables are mapped to, and so it is well defined what they are mapped to after the replacement.

It will also be clear that this unifier u' will be equally or or more general than the original unifier u, since we obtain e_3 again if we follow it with the substitution that replaced each new variable with the fragment it replaced.

The number of equivalence classes of unifiers that are generated by the previous process can be shown to be finite. To show this, we introduce the concept of *symbol ordering*. By this we mean a linear order over the keys and bracket occurrences in e_1 and e_2 that (1) allows occurrences from e_1 to be merged with occurrences from e_2 if they concern the same symbol and (2) respects the linear order of the occurrences in e_1 and e_2. As an example of a symbol ordering consider the following ordering, where the central horizontal line indicates the linear order. Here the solid lines indicate the ordering defined by e_1 and e_2 in the previous example, and the dahsed edges indicated the added orderings to make it linear.
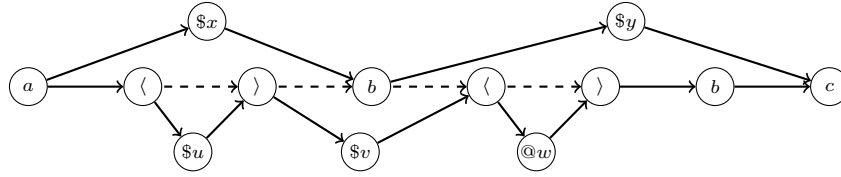
It is easy to see that every generated diagram for $e_1 = e_2$ will define a symbol ordering in its central horizontal line. Moreover, the symbol ordering, along with the original linear order within e_1 and e_2, completely determines the diagram since all that is required is to select fresh variables for the dashed edges. This implies that this also determines then the unifier it defines. Since the linear orders in e_1 and e_2 can only be combined into a symbol ordering in finitely many ways, it follows that there are only a finite number of distinct (up to the choice of the fresh identifiers) unifiers that are generated by the described process for generalising unifiers. $\square$

The previous result allows us to show that we can remove equations from sets of equationally acyclic rules.

Lemma 5.18. *Every J-Logic rule that is equationally acyclic is equivalent to a finite set of equality-free rules. Also, every jaegd where we would allow equalities in the body such that it is equationally acyclic, is equivalent to a finite set of equality-free jaegds.*

Proof. We show by induction that a equationally acyclic rule with $n > 0$ equations, can be rewritten to an equivalent set of equationally acyclic rules with $n - 1$ equations.

Let us consider a rule with equation $e_1 = e_2$. By Lemma 5.17 we know that there is a finite set of equivalence classes of most-general unifiers of e_1 and e_2. We can select for each equivalence class a representative that maps variables to path expression with only fresh variables.

From the initial rule with the equation $e_1 = e_2$ we generate now a set of rules by (1) removing this equality and (2) generate a rule for each unifier in the set of unifiers by applying it to the remainder of the rule. Recall that a valuation, a function that maps atomic variables to atomic keys and path variables to paths, satisfies $e_1 = e_2$ iff it maps e_1 and e_2 to the same path. It follows that this holds iff the valuation is a unifier of e_1 and e_2, which in turn holds iff the valuation is an equal or less general unifier than one of the most-general unifiers. It follows that replacing the initial rule with the generated set of rules does not change the semantics.

As a final step we show that the resulting set of rules remains equationally acyclic. Let us consider one of the newly generated rules, and assume it was generated by the unifier u. Assume

that the application of u caused a cycle in the equation graph of the generated rule. We can them map this cycle back to a cycle that existed in the equation graph of the initial rule as follows:

- Consider an edge between two variables caused by occurrences that already existed before the application of u. Then the corresponding edge already existed in the equation graph of the initial rule.

- Consider an edge between an old variable x and a new variable y added by u. Let z be the unique variable that was replaced with a path expression containing y. Then, an edge between z and x already existed in the equation graph of the initial rule.

- Consider an edge between new variable x and new variable y. Let v and w be the unique variables that were replaced to introduce x and y, respectively. If v and w are on opposite sides in $e_1 = e_2$ then there is a corresponding edge between v and w in the old equation graph. If v and w are on the same side, then there must be a variable v' with which v overlapped to generate x and which is on the other side than v. It follows that there is an edge between v and v', and between v' and w in the old equation graph.

In all considered cases it holds that for every edge in the new equation graph there is a corresponding edge or path in the old equation graph if we map new variables back to the old variable that generated them. It follows that for every cycle in the new equation graph there must have already been a corresponding cycle in the old equation graph. $\square$

We are now ready for the

Proof of Theorem 5.9. Let $\mathbf{P}$ be an equationally acyclic program computing a query Q from $\mathcal{V}_{\text{in}}$ to $\mathcal{V}_{\text{out}}$. For the sake of simplicity we assume $\mathcal{V}_{\text{in}} = \{R\}$ and $\mathcal{V}_{\text{out}} = \{S\}$ consist of a single relation name. Then $\mathbf{P}$ is a set of rules with S in the head predicate and R as the only EDB relation. By Lemma 5.18, we can transform $\mathbf{P}$ into a program without equalities, so we will assume from here on that $\mathbf{P}$ contains no equalities.

Recall from Example 5.10 the set of six jaegds $\Delta = \{\delta_1, \ldots, \delta_6\}$ that expresses properness. The main idea is that Q has the object–object property if and only if the "Δ–Δ implication problem" holds for Q. We then leverage the observation made in Example 5.15 that chasing from Δ is unambiguous.

More precisely, for a relation name P and each $i = 1, \ldots, 6$, let δ_i^P be the version of δ_i where we substitute P for the name D. Let $\Delta^R = \{\delta_1^R, \ldots, \delta_6^R\}$. Then for each each $i = 1, \ldots, 6$ and every instance $I \models \Delta^R$, we want to check that $Q(I) \models \delta_i^S$.

Let us begin with δ_1^S. We consider every pair of rules $(\mathbf{r}_1, \mathbf{r}_2)$ from $\mathbf{P}$, where $\mathbf{r}_1$ and $\mathbf{r}_2$ can also be the same rule. Let the head of $\mathbf{r}_j$ be $S(e_j : t_j)$, for $j = 1, 2$. We apply a variable renaming ρ so that $\mathbf{r}_1$ and $\rho(\mathbf{r}_2)$ have no variables in common. Now construct a jaegd with equalities from δ_1, $\mathbf{r}_1$ and $\rho(\mathbf{r}_2)$ as follows. Using fresh variables $\$x$, $@i$ and $@j$, the body consists of the bodies of $\mathbf{r}_1$ and $\rho(\mathbf{r}_2)$, together with the equalities $\$x = e_1$, $\$x = \rho(e_2)$, $t_1 = @i$, and $\rho(t_2) = @j$. The head is $(@i = @j)$.

Since the rules in $\mathbf{P}$ contain no equalities, and every head contains each variable at most once, it follows that the constructed rule is equationally acyclic. I follows by Lemma 5.18, that this jaegd with equalities is equivalent to a finite set of jaegds, which we denote by $\Delta_1^{\mathbf{r}_1, \mathbf{r}_2}$. It is now clear that $Q(I) \models \delta_1^S$ for every $I \models \Delta^R$, if and only if every jaegd in $\Delta_1^{\mathbf{r}_1, \mathbf{r}_2}$ is logically implied by Δ^R. This is a unambiguous case of the implication problem, so it can be solved by the chase.

Checking implication for δ_2–δ_5 is similar. For example, from δ_3, $\mathbf{r}_1$ and $\rho(\mathbf{r}_2)$ and fresh variables $\$x$, $\$y$ and $@i$, we construct a denial constraint with equalities having as body the bodies of $\mathbf{r}_1$ and $\rho(\mathbf{r}_2)$ together with the equalities $e_1 = \$x$, $t_1 = @i$, and $\rho(e_2) = \$x.\$y$. $\qquad\square$

Computational complexity Like the implication problem for egds in the relational model, the computational complexity of the unambiguous cases of the implication problem for jaegds is NP-complete. Note, however, that in the above proof we only need to chase jaegds from $\Delta_i^{\mathbf{r}_1,\mathbf{r}_2}$ with the fixed set of jaegds Δ^R. Hence each application of the chase would be polynomial, were it not for the following caveat. The caveat is that $\Delta_i^{\mathbf{r}_1,\mathbf{r}_2}$ is obtained after elimination of equalities, which can result in exponentially many rules, and these rules may be exponential in size due to the repeated doubling. Even when the given program has no equalities, there are still equalities to be eliminated in the jaegd constructed from $\mathbf{r}_1$ and $\mathbf{r}_2$. We thus can only conclude an exponential-time upper bound on the complexity of the object–object problem for positive nonrecursive J-Logic programs. We leave the exact complexity open.

6 The containment problem over flat instances

Let $\mathbf{P}_1$ and $\mathbf{P}_2$ be J-Logic programs both expressing a query from $\mathcal{V}_{\text{in}}$ to $\mathcal{V}_{\text{out}}$; let Q_j be the query expressed by $\mathbf{P}_j$.

Let $\mathcal{F}$ be a family of instances. The *containment problem over* $\mathcal{F}$ asks, given $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathcal{V}_{\text{in}}$ and $\mathcal{V}_{\text{out}}$ as above, whether $Q_1(I) \subseteq Q_2(I)$ for all instances I over $\mathcal{V}_{\text{in}}$ belonging to $\mathcal{F}$. Recall that an instance is *flat* if no packed keys occur in it. In this section we show:

Theorem 6.1. *Let $\mathcal{F}$ be the set of flat instances, and let $\mathcal{PF}$ be the set of proper flat instances. For positive nonrecursive programs, containment over $\mathcal{F}$ is decidable, and so is containment over $\mathcal{PF}$.*

Note that we restrict attention to flat instances. Indeed, our current solution does not work with packed keys in the inputs (see Remark 6.5). It is an interesting topic for further research to see whether our our solution can be extended in the presence of packing.

To solve the containment problem over $\mathcal{F}$ one can take inspiration from the inclusion problem for *pattern languages* over an infinite alphabet [19]. The main additional aspect here is the distinction between atomic variables and path variables.

In the field of pattern languages, a pattern is a finite sequence of constants and path variables, so, in our terminology, a path expression without atomic variables and packed key expressions. The *language* of a pattern e is the set $L(e)$ of all flat paths p for which there exists a valuation h such that $h(e) = p$. Here, a flat path is a path in which no packed keys occur, i.e., a nonempty sequence of atomic keys. Note that this essentially interprets patterns over an infinite alphabet, since our universe **dom** of atomic keys is infinite. In this case it is known [19] that $L(e_1) \subseteq L(e_2)$ if and only if there exists a variable mapping h such that $h(e_2) = e_1$. When atomic variables come into play, however, this "homomorphism property" is no longer necessary for containment.

Example 6.2. Let us allow atomic variables in patterns. Then consider the following four patterns:

$$e_1 = \$x.\$y \qquad\qquad e_3 = @x.\$y.@z$$
$$e_2 = @x.\$y \qquad\qquad e_4 = \$u.@v.\$w$$

Then e_1 and e_2 describe the same language, namely all flat paths of length at least two. There is a variable mapping from e_1 to e_2 but not from e_2 to e_1, since a variable mapping cannot map an atomic variable (in this case $@x$) to a path variable (in this case $\$x$). Also e_3 and e_4 describe the same language, namely all flat paths of length at least three. Here there is neither a variable mapping from e_3 to e_4 nor one from e_4 to e_3.

The simplistic idea to just allow weak variable mappings does not work. For example, there is a weak variable mapping from $@x$ to $\$x$ but $L(\$x)$ is not contained in $L(@x)$. ☐

We next develop our general solution to the containment problem over flat instances. For simplicity, we always consider positive nonrecursive programs expressing a query from $\mathcal{V}_{\text{in}}$ to $\mathcal{V}_{\text{out}}$ where $\mathcal{V}_{\text{out}} = \{S\}$ is a single-relation vocabulary. Such programs can be written as finite sets of rules with S in the head predicate and relation names from $\mathcal{V}_{\text{in}}$ in the bodies.

It is sufficient to solve the containment problem given programs $\mathbf{P}_1$ and $\mathbf{P}_2$ where $\mathbf{P}_1$ consists of a single rule $\mathbf{r}_1$ (since otherwies we can check containment for all rules of $\mathbf{P}_1$ separately). Moreover, we can make the following proviso:

Proviso. *The body of $\mathbf{r}_1$ and the bodies of rules in $\mathbf{P}_2$ do not have equalities. Moreover, these bodies are flat, i.e., do not use packing.*

The first part of the proviso is justified by Lemma 5.18. The second part is justified because we work over flat instances: non-flat bodies can never match anyway. The heads may still use packing.

We begin by noting that there *is* a simple homomorphism theorem when $\mathbf{r}_1$ does not have path variables.

Proposition 6.3. *Assume $\mathbf{r}_1$ does not have path variables. Then $\mathbf{r}_1$ is contained in $\mathbf{P}_2$ over $\mathcal{F}$ if and only if there exists a rule $\mathbf{r}_2 \in \mathbf{P}_2$ such that there is a homomorphism from B_2 to B_1, mapping H_2 to H_1. Here, B_i and H_i denote the body and the head of $\mathbf{r}_i$.*

Proof. The if-direction is straightforward. For the only-if direction, we view B_1 as a (flat) instance I by viewing each variable as an atomic key (called a frozen variable). We similarly view H_1 as a fact. Then clearly $H_1 \in \mathbf{r}_1(I)$, so also $H_1 \in \mathbf{P}_2(I)$. Hence there exists $\mathbf{r}_2 \in \mathbf{P}_2$ and a valuation ν such that $\nu(B_2) \subseteq B_1$ and $\nu(H_2) = H_1$. Since $\mathbf{r}_1$ does not have path variables, ν can map atomic variables only to constants or to (frozen) atomic variables. Hence, we can view ν as a homomorphism from $\mathbf{r}_2$ to $\mathbf{r}_1$. ☐

We now reduce the containment problem where $\mathbf{r}_1$ has path variables, to infinitely many calls to the containment problem where $\mathbf{r}_1$ does not have path variables. Thereto, we associate to every path variable $\$x$ an infinite sequence $@x^1$, $@x^2$, ... of atomic variables. Obviously, for distinct path variables $\$x$ and $\$y$ we assume $@x^i$ and $@y^j$ are distinct for all i and j.

A *variant* of $\mathbf{r}_1$ is a rule obtained from $\mathbf{r}_1$ as follows. For every path variable $\$x$ in $\mathbf{r}_1$, choose a natural number $n_{\$x}$. We call $n_{\$x}$ the *chosen length* for $\$x$. Now replace each occurrence of $\$x$ in $\mathbf{r}_1$ by the sequence $@x^1 \ldots @x^{n_{\$x}}$. Thus, as soon as $\mathbf{r}_1$ has at least one path variable, there are infinitely many variants of $\mathbf{r}_1$.

The following is now clear:

Proposition 6.4. *$\mathbf{r}_1$ is equivalent, over $\mathcal{F}$, to the infinite union of its variants. In particular, $\mathbf{r}_1$ is contained in $\mathbf{P}_2$ over $\mathcal{F}$ if and only if every variant of $\mathbf{r}_1$ is contained in $\mathbf{P}_2$ over $\mathcal{F}$.*

Remark 6.5. The above proposition only works over flat instances. Consider, for example, the rules

$$\mathbf{r}_2 = S(c : \emptyset) \leftarrow R(@u.\$z : \emptyset)$$
$$\mathbf{r}_1 = S(c : \emptyset) \leftarrow R(\$x.\$y : \emptyset)$$

Rule $\mathbf{r}_2$ tests if R contains a path of length at least two, starting with an atomic key, and with the empty value at the leaf. If so, the fact $S(c : \emptyset)$ is returned (c is some constant). An example of a variant of $\mathbf{r}_1$, with 2 as chosen length for $\$x$ and 3 for $\$y$, is

$$S(c : \emptyset) \leftarrow R(@x^1.@x^2.@y^1.@y^2.@y^3 : \emptyset).$$

We see that this variant, and indeed every variant, of $\mathbf{r}_1$ is contained in $\mathbf{r}_2$. Nevertheless $\mathbf{r}_1$ is not contained in $\mathbf{r}_2$ over all instances, as witnessed by the instance $I = \{R(\langle a \rangle.b : \emptyset)\}$. $\qquad\square$

The above proposition gives us infinitely many variant containments to check. Our final step reduces this to a finite number.

Proposition 6.6. *Let m be the number of atomic variables used in $\mathbf{P}_2$. Assume all variants of $\mathbf{r}_1$ with chosen lengths up to $m + 1$ are contained in $\mathbf{P}_2$ over $\mathcal{F}$. Then every variant of $\mathbf{r}_1$ is contained in $\mathbf{P}_2$ over $\mathcal{F}$.*

Proof. By Proposition 6.3, it is sufficient to show the following claim. *Let $\mathbf{r}$ be a variant of $\mathbf{r}_1$ with a chosen length $k \geq m + 1$ for some path variable $\$x$. Assume there is a homomorphism h from a rule $\mathbf{r}_2 \in \mathbf{P}_2$ to $\mathbf{r}$. Let $\mathbf{r}'$ be the same variant as $\mathbf{r}$, except that the chosen length for $\$x$ is increased to $k' > k$. Then there is still a homomorphism from $\mathbf{r}_2$ to $\mathbf{r}'$.*

We argue the claim as follows. Since $k > m$, one of the variant variables for $\$x$, say $@x^j$, is not in the image of h applied to any atomic variable from $\mathbf{r}_2$. Hence it only occurs in the images of some path variables. Each such path variable is mapped by h to a flat path expression in which $@x^j$ occurs. Now for each such variable $\$z$, modify $h(\$z)$ by inserting the sequence $@x^{k+1} \ldots @x^{k'}$ behind each occurrence of $@x^j$. The resulting variable mapping h' gives us the desired homomorphism from $\mathbf{r}_2$ to $\mathbf{r}'$. (The only detail is that the variant sequence for $\$x$ is permuted a bit, instead of $@x^1 \ldots @x^{k'}$ it is now $@x^1 \ldots @x^j @x^{k+1} \ldots @x^{k'} @x^{j+1} \ldots @x^k$.) $\qquad\square$

We conclude that containment of $\mathbf{r}_1$ in $\mathbf{P}_2$ over $\mathcal{F}$ is decidable with Π_2^P complexity. Indeed, instead of trying all variants of $\mathbf{r}_1$, as given by Proposition 6.4, it suffices to try all variants of $\mathbf{r}_1$ with chosen length bounds as given by Proposition 6.6. For each variant we test the existence of a homomorphism as given by Proposition 6.3. We leave open whether the problem is actually Π_2^P-hard.

Containment over proper flat instances For simplicity, let us assume that $\mathcal{V}_{in}$ consists of a single relation name D. Recall from Example 5.10 the set Δ of jaegds that expresses properness. We can chase a rule with Δ in much the same way as we chase a jaegd as defined in Section 5.1. We establish:

Proposition 6.7. *$\mathbf{r}_1$ is contained in $\mathbf{P}_2$ over $\mathcal{PF}$ if and only if either chasing $\mathbf{r}_1$ with Δ fails, or it succeeds and results in a rule $\mathbf{r}$ such that $\mathbf{r}$ is contained in $\mathbf{P}_2$ over $\mathcal{F}$.*

Proof. For the if-direction, first assume the chase fails. Then $\mathbf{r}_1(I)$ is empty on all proper instances so containment holds trivially. Next assume the chase succeeds and results in the rule $\mathbf{r}$. Let I be

a proper flat instance. By Lemma 5.12, appropriately adapted to rules, we have $\mathbf{r}_1(I) = \mathbf{r}(I)$. By the given, $\mathbf{r}(I) \subseteq \mathbf{P}_2(I)$ and we are done.

For the only-if direction, suppose the chase succeeds and results in the rule $\mathbf{r} = H \leftarrow B$. By Proposition 6.4, we have to show that every variant $\mathbf{r}' = H' \leftarrow B'$ of $\mathbf{r}$ is contained in $\mathbf{P}_2$. We can view B as an instance I by viewing each variable as an atomic key (I is called a frozen body). Because chasing from Δ is unambiguous, we obtain as in the proof of Theorem 5.16 that $I \models \Delta$, i.e., I is proper.

But then I', the frozen variant body B', is also proper. Indeed, by replacing each frozen path variable by a sequence of frozen atomic variables, the functional dependency from paths to atomic values remains satisfied. Moreover, I' is also still prefix-free. In proof, suppose $D(p' : v) \in I'$. The last element s of p' is either a constant or a frozen atomic variable from $\mathbf{r}$, or a frozen atomic variable $@x^n$ coming from a path variable $\$x$ in $\mathbf{r}$. In the latter case, n must be the chosen length for $\$x$. Now suppose there would exists $D(p'.q' : u) \in I'$. Since the first symbol of q' follows the last symbol of p', it is either again a constant or frozen atomic variable from $\mathbf{r}$, or a frozen atomic variable y^1 coming from a path variable $\$u$ in $\mathbf{r}$. We conclude that the presence of $D(p' : v)$ and $D(p'.q' : u)$ in I' would imply the presence of some $D(p : v)$ and $D(p.q : u)$ in I, which is impossible because I is prefix-free.

Clearly, $H' \in \mathbf{r}'(I')$, so also $H' \in \mathbf{r}(I')$ since $\mathbf{r}'$ is a variant of $\mathbf{r}$. Furthermore, since $\mathbf{r}$ was obtained from $\mathbf{r}_1$ by applying chase steps, which are applications of homomorphisms, also $H' \in \mathbf{r}_1(I')$. By the given, then $H' \in \mathbf{P}_2(I')$. This means there exists $\mathbf{r}_2 \in \mathbf{P}_2$ and a matching $\nu : B_2 \to I'$ such that $\nu(B_2) \subseteq I'$ and $\nu(H_2) = H'$. Since $\mathbf{r}'$ does not have path variables, ν can be viewed as a homomorphism from $\mathbf{r}_2$ to $\mathbf{r}'$. Hence, $\mathbf{r}'$ is contained in $\mathbf{P}_2$ as desired. $\quad\square$

7 Conclusion

Thanks to the deterministic nature of JSON objects, it is very convenient to view objects as sets of key sequences paired with atomic values. We recommend the use of path variables, ranging over key sequences, in languages for JSON querying for accessing deeply nested data. While the data complexity is polynomial-time, it would be interesting to investigate practical query processing issues involving path variables.

Furthermore, packing is a versatile tool not only for expressive power and the generation of new keys, but also for marking parts of sequences, duplicate elimination, and other tricks. We recommend that practical JSON query processors support packed keys.

Our technical results have shown that the proposed approach is workable. Much further work can be done: Is there a nonrecursive flat–flat theorem? What is the exact complexity of the object–object problem for nonrecursive programs? Is the containment problem for nonrecursive programs decidable in the presence of packing? How does the complexity of the containment problem change when equalities are allowed in rules?

During our research we also encountered the following intriguing puzzle. Consider the extreme case where there exists only one atomic key, and there is no packing. Then J-Logic amounts to monadic Datalog with stratified negation over sets of sequences of a's, with path variables and atomic variables. This corresponds to monadic Datalog with stratified negation over sets of natural numbers, with natural number constants and variables, and addition as the only operation. Which functions on sets of natural numbers are expressible in this language?

References

[1] S. Abiteboul and R. Hull. Data functions, datalog and negation. In H. Boral and P.A. Larson, editors, *1988 Proceedings SIGMOD International Conference on Management of Data*, pages 143–153. ACM Press, 1988.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45(5):798–842, 1998.

[4] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 1990.

[5] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[6] P.C. Arocena, B. Glavic, and R.J. Miller. Value invention in data exchange. In *Proceedings 2013 SIGMOD Conference*, pages 157–168. ACM, 2013.

[7] P. Barceló and R. Pichler, editors. *Datalog in Academia and Industry: Second International Workshop, Datalog 2.0*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012.

[8] C. Beeri and M.Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[9] A. Bonner and G. Mecca. Sequences, Datalog, and transducers. *Journal of Computer and System Sciences*, 57:234–259, 1998.

[10] A.J. Bonner and G. Mecca. Querying sequence databases with transducers. *Acta Informatica*, 36:511–544, 2000.

[11] P. Buneman, A. Deutsch, and W.-C. Tan. A deterministic model for semi-structured data. `http://users.soe.ucsc.edu/~tan/papers/1998/icdt.pdf`. Presented at the Workshop on Query Processing for Semistructured Data and Non-standard Data Formats, Jerusalem, Israel, January 13, 1999.

[12] P. Buneman, S.A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[13] L. Cabibbo. The expressive power of stratified logic programs with value invention. *Information and Computation*, 147(1):22–56, 1998.

[14] A.K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[15] O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors. *Datalog Reloaded: First International Workshop, Datalog 2010*, volume 6702 of *Lecture Notes in Computer Science*. Springer, 2011.

[16] The JSON data interchange format. Standard ECMA-404, October 2013.

[17] D. Florescu and G. Fourny. JSONiq: The history of a query language. *IEEE Internet Computing*, 17(5):86–90, 2013.

[18] G. Fourny. JSONiq, the SQL of NoSQL. `http://www.28.io/jsoniq-the-sql-of-nosql`. Retrieved 25 November 2016.

[19] D.D. Freydenberger and D. Reidenbach. Bad news on decision problems for patterns. *Information and Computation*, 208(1):83–96, 2010.

[20] T. Furche, G. Gottlob, B. Neumayr, and E. Sallinger. Data wrangling for big data: Towards a lingua franca for data wrangling. In R. Pichler and A. Soares da Silva, editors, *Proceedings 10th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 1644 of *CEUR Workshop Proceedings*, 2016.

[21] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[22] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994.

[23] Jan Hidders, Jan Paredaens, and Jan Van den Bussche. J-logic: Logical foundations for json querying. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, page 137–149, New York, NY, USA, 2017. Association for Computing Machinery.

[24] S.S. Huang, T.J. Green, and B.T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings 2011 ACM SIGMOD International Conference on Management of Data*, pages 1213–1216. ACM Press, 2011.

[25] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 455–468. Morgan Kaufmann, 1990.

[26] M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 47(1):77–120, 1993.

[27] A. Klug and R. Price. Determining view dependencies using tableaux. *ACM Transactions on Database Systems*, 7:361–380, 1982.

[28] G. Kuper and M. Vardi. The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.

[29] G. Mecca and A.J. Bonner. Query languages for sequence databases: Termination and complexity. *IEEE Transactions on Knowledge and Data Engineering*, 13(3):519–525, 2001.

[30] K.W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ query language: Configurable, unifying and semi-structured. arXiv:1405.3631, 2015.

[31] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems*, 17(1):65–93, 1992.

[32] F. Pezoa, J.L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In *Proceedings 25th International Conference on World Wide Web*, pages 263–273, 2016.

[33] A. Poggi et al. Linking data to ontologies. *Journal on Data Semantics*, 10:133–173, 2008.

[34] K. Tajima. Schemaless semistructured data revisited: Reinventing Peter Buneman's deterministic semistructured data model. In V. Tannen, L. Wong, et al., editors, *In Search of Elegance in the Theory and Practice of Computation*, volume 8000 of *Lecture Notes in Computer Science*, pages 466–482. Springer, 2013.

[35] J. Van den Bussche and J. Paredaens. The expressive power of complex values in object-based data models. *Information and Computation*, 120:220–236, 1995.

[36] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.

[37] T.L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proceedings 17th International Conference on Database Theory*, pages 96–106, 2014.