JSON: Data model, Query languages and Schema specification

Pierre Bourhis
CNRS CRIStAL UMR9189, Lille; INRIA Lille
pierre.bourhis@univ-lille1.fr

Juan L. Reutter
PUC Chile and Center for Semantic Web Research
jreutter@ing.puc.cl

Fernando Suárez
PUC Chile and Center for Semantic Web Research
fsuarez1@uc.cl

Domagoj Vrgoč
PUC Chile and Center for Semantic Web Research
dvrgoc@ing.puc.cl

ABSTRACT

Despite the fact that JSON is currently one of the most popular formats for exchanging data on the Web, there are very few studies on this topic and there is no agreement upon a theoretical framework for dealing with JSON. Therefore in this paper we propose a formal data model for JSON documents and, based on the common features present in available systems using JSON, we define a lightweight query language allowing us to navigate through JSON documents. We also introduce a logic capturing the schema proposal for JSON and study the complexity of basic computational tasks associated with these two formalisms.

CCS Concepts

•**Information systems** → **Database design and models; Query languages;** *Web data description languages;*
•**Theory of computation** → *Data modeling;*

Keywords

JSON; Schema languages; Navigation

1. INTRODUCTION

JavaScript Object Notation (JSON) [17, 11] is a lightweight format based on the data types of the JavaScript programming language. In their essence, JSON documents are dictionaries consisting of key-value pairs, where the value can again be a JSON document, thus allowing an arbitrary level of nesting. An example of a JSON document is given in Figure 1. As we can see here, apart from simple dictionaries, JSON also supports arrays and atomic types such as integers and strings. Arrays and dictionaries can again contain arbitrary JSON documents, thus making the format fully compositional.

```
{
  "name": {
     "first": "John",
     "last": "Doe"
     },
  "age": 32,
  "hobbies": ["fishing","yoga"]
}
```

Figure 1: A simple JSON document.

Due to its simplicity, and the fact that it is easily readable both by humans and by machines, JSON is quickly becoming one of the most popular formats for exchanging data on the Web. This is particularly evident with Web services communicating with their users through an Application Programming Interface (API), as JSON is currently the predominant format for sending API requests and responses over the HTTP protocol. Additionally, JSON format is much used in database systems built around the NoSQL paradigm (see e.g. [22, 2, 26]), or graph databases (see e.g. [32]).

Despite its popularity, the coverage of the specifics of JSON format in the research literature is very sparse, and to the best of our knowledge, there is still no agreement on the correct data model for JSON, no formalisation of the core query features which JSON systems should support, nor a logical foundation for JSON Schema specification. And while some preliminary studies do exist [23, 21, 8, 28], as far as we are aware, no attempt to describe a theoretical basis for JSON has been made by the research community. Therefore, the main objective of this paper is to formally define an appropriate data model for JSON, identify the key querying features provided by the existing JSON systems, and to propose a logic allowing us to specify schema constraints for JSON documents.

In order to define the data model, we examine the key characteristics of JSON documents and how they are used in practice. As a result we obtain a tree-shaped structure very similar to the ordered data-tree model of XML [7], but with some key differences. The first difference is that JSON trees are deterministic by design, as each key can appear at most once inside a dictionary. This has various implications at the time of querying JSON documents: on one hand we sometimes deal with languages far simpler than XML, but on the other hand this key restriction can make static analy-

PODS'17, May 14-19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4198-1/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3034786.3056120

sis more complicated, even for simpler queries. Next, arrays are explicitly present in JSON, which is not the case in XML. Of course, the ordered structure of XML could be used to simulate arrays, but the defining feature of each JSON dictionary is that it is unordered, thus dictating the nodes of our tree to be typed accordingly. And finally, JSON values are again JSON objects, thus making equality comparisons much more complex than in case of XML, since we are now comparing subtrees, and not just atomic values. We cover all of these features of JSON in more detail in the paper, and we also argue that, while technically possible (albeit, in a very awkward manner), coding JSON documents using XML might not be the best solution in practice.

Next, we consider the problem of querying JSON. As there is currently no agreed upon query language in place, we examine an array of practical JSON systems, ranging from programming languages such as Python [13], to fully operational JSON databases such as MongoDB [22], and isolate what we consider to be key concepts for accessing JSON documents. As we will see, the main focus in many systems is on navigating the structure of a JSON tree, therefore we propose a navigational logic for JSON documents based on similar approaches from the realm of XML [12], or graph databases [3, 20]. We then show how our logic captures common use cases for JSON, extend it with additional features, and demonstrate that it respects the "lightweight nature" of the JSON format, since it can be evaluated very efficiently, and it also has reasonable complexity of main static tasks. Interestingly, sometimes we can reuse results devised for other similar languages such as XPath or Propositional Dynamic Logic, but the nature of JSON and the functionalities present in query languages also demand new approaches or a refinement of these techniques.

Another important aspect of working with any data format is being able to specify the structure of documents. A usual way to do this is through schema specification, and in the case of JSON, there is indeed a draft proposal for a schema language [18] (called JSON Schema), which has recently been formalised in [28]. Based on the formalisation of [28] we define a logic capturing the full formal specification of JSON Schema, show that it is essentially equivalent to the navigational language we propose for querying JSON, and study the complexity of its evaluation and static tasks. However, once we take into account the recursive functionalities of JSON Schema, we arrive at a powerful new formalism that is much more difficult to encompass inside well known frameworks.

Finally, since theoretical study of JSON is still in its early stages, we close with a series of open problems and directions for future research.

Organisation. We formally define JSON and some of its features in Section 2. The appropriate data model for JSON is discussed in Section 3, and its query language in Section 4. In Section 5 we define a logic capturing a schema specification for JSON. Our conclusions and the directions for future work are discussed in Section 6. Due to the lack of space most proofs are placed in the appendix to this paper.

2. PRELIMINARIES

JSON documents. We start by fixing some notation regarding JSON documents. The full JSON specification defines seven types of values: objects, arrays, strings, numbers and the values true, false and null [9]. However, to abstract from encoding details we assume our JSON documents are only formed by objects, arrays, strings and natural numbers.

Formally, denote by Σ the set of all unicode characters. JSON values are defined as follows. First, any natural number $n \geqslant 0$ is a JSON value, called a *number*. Furthermore, if `s` is a string in Σ^*, then `"s"` is a JSON value, called a *string*. Next, if $v_1, \dots, v_n$ are JSON values and $s_1, \dots, s_n$ are pairwise distinct string values, then $\{s_1 : v_1, \dots, s_n : v_n\}$ is a JSON value, called an *object*. In this case, each $s_i : v_i$ is called a key-value pair of this object. Finally, if $v_1, \dots, v_n$ are JSON values then $[v_1, \dots, v_n]$ is a JSON value called an *array*. In this case $v_1, \dots, v_n$ are called the *elements* of the array. Note that in the case of arrays and objects the values v_i can again be objects or arrays, thus allowing the documents an arbitrary level of nesting. A *JSON document* (or just document) is any JSON value. From now on we will use the term JSON document and JSON value interchangeably.

JSON navigation instructions. Arguably all systems using JSON base the extraction of information in what we call *JSON navigation instructions*. The notation used to specify JSON navigation instructions varies from system to system, but it always follows the same two principles:

- If J is a JSON object, then one should be able to access the JSON value in a specific key-value pair of this object.
- If J is a JSON array, then one should be able to access the i-th element of J.

In this paper we adopt the python notation for navigation instructions: If J is an object, then $J[key]$ is the value of J whose key is the string value `"key"`. Likewise, if J is an array, then $J[n]$, for a natural number n, contains the n-th element of J[1].

As far as we are aware, all JSON systems use JSON navigation instructions as a primitive for querying JSON documents, and in particular it is so for the systems we have reviewed in detail: Python and other programming languages [13], the mongoDB database [22], the JSON Path query language [14] and the SQL++ project that tries to bridge relational and JSON databases [24].

At this point it is important to note a few important consequences of using JSON navigation instructions, as these have an important role at the time of formalising this framework.

First, note that we do not have a way of obtaining the keys of JSON objects. For example, if `J` is the object `{"first":"John", "last":"Doe"}` we could issue the instructions $J[\mathit{first}]$ to obtain the value of the pair `"first":"John"`, which is the string value `"John"`. Or use $J[\mathit{last}]$ to obtain the value of the pair `"last":"Doe"`. However, there is no instruction that can retrieve the keys inside this document (i.e. `"first"` and `"last"` in this case).

Similarly, for the case of the arrays, the access is essentially a *random access*: we can access the i-th element of an array, and most of the time there are primitives to access the first or the last element of arrays. However, we can not reason about different elements of the array. For example, for the array K = `[12,5,22]` we can not retrieve, say, an element (or any element) which is greater than the first element of K. Some systems do feature FLWR expressions that

[1]Some JSON systems prefer using a *dot* notation, where J.key and J.n are the equivalents of J[key] and $J[n]$.

support iterating over all elements. But this iteration is itself treated as a series of random accesses, using commands such as `For i in (0,n) print(J[i])`.

3. DATA MODEL FOR JSON

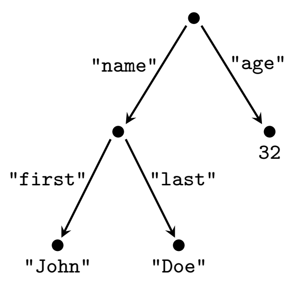
In this section we propose a formal data model for JSON documents whose goal is to closely reflect the manner in which JSON is manipulated using JSON navigation instructions, and that will be used later on as the basis of our formalisation of JSON query and schema languages. We begin by introducing our formal model, called JSON trees. Afterwards we discuss the main differences between JSON trees and other well-studied tree formalisms such as data trees or XML.

3.1 JSON trees

JSON objects are by definition compositional: each JSON object is a set of key-value pairs, in which values can again be JSON objects. This naturally suggests using a tree-shaped structure to model JSON documents. However, this structure must preserve the compositional nature of JSON. That is, if each node of the tree structure represents a JSON document, then the children of each node must represent the documents nested within it. For instance, consider the following JSON document J.

```
{
  "name": {
     "first": "John",
     "last": "Doe"
     },
  "age": 32
}
```

as explained before, this document is a JSON *object* which contains two keys: `"name"` and `"age"`. Furthermore, the value of the key `"name"` is another JSON document and the value of the key `"age"` is the integer 32. There are in total 5 JSON values inside this object: the complete document itself, plus the literals `32`, `"John"` and `"Doe"`, and the object `"name": {"first":"John", "last":"Doe"}`. So how should a tree representation of the document J look like? If we are to preserve the compositional structure of JSON, then the most natural representation is by using the following edge-labelled tree:



The root of tree represents the entire document. The two edges labelled `"name"` and `"age"` represent two keys inside this JSON object, and they lead to nodes representing their respective values. In the case of the key `"age"` this is just an integer, while in the case of `"name"` we obtain another JSON object that is represented as a subtree of the entire tree.

Finally, we need to enforce the property of JSON that no object can have two keys with the same name, thus making the model deterministic in some sense, since each node will have only one child reachable by an edge with a specific label. Let us briefly summarise the properties of our model so far.

Labelled edges. Edges in our model are labelled by the keys forming the key-value pairs of objects. This means that we can directly follow the label of edges when issuing JSON navigation instructions, and also means that information of keys is represented in a different medium than JSON values (labels for the former, nodes for the latter). This is inline with the way JSON navigation instructions work, as one can only retrieve values of key-value pairs, but not the keys themselves. To comply with the JSON standard, we disallow trees where a same edge label is repeated in two different edges leaving a node.

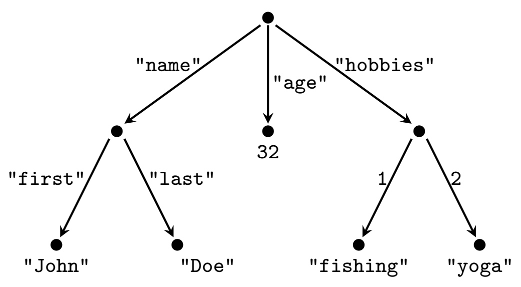
Compositional structure. One of the advantages of our tree representation is that any of its subtrees represent a JSON document themselves. In fact, the five possible subtrees of the tree above correspond to the five JSON values present in the JSON J.

Atomic values. Finally, some elements of a JSON document are actual values, such as integers or strings. For this reason leaf nodes corresponding to integers and strings will also be assigned a value they carry. Leaf nodes without a value represent empty objects: that is, documents of the form `{}`.

Although this model is simple and conceptually clear, we are missing a way of representing arrays. Indeed, consider again the document from Figure 1 (call this document J_2). In J_2 the value of the key `"hobbies"` is an array: another feature explicitly present in JSON that thus needs to be reflected in our model.

As arrays are ordered, this might suggest that we can have some nodes whose children form an ordered list of siblings, much like in the case of XML. But this would not be conceptually correct, for the following two reasons. First, as we have explained, JSON navigation instructions use random access to access elements in arrays. For example, the navigation instruction used to retrieve an element of an array is of the form $J_2[hobbies][i]$, aimed at obtaining the i-th element of the array under the key `"hobbies"`. But more importantly, we do not want to treat arrays as a list because lists naturally suggest navigating through different elements of the list. On the contrary, none of the systems we reviewed feature a way of navigating from one element of the array to another element. That is, once we retrieve the first element of the array under the key `"hobbies"`, we have no way of linking it to its siblings.

We choose to model JSON arrays as nodes whose children are accessed by axes labelled with natural numbers reflecting their position in the array. Namely, in the case of JSON document J_2 above we obtain the following representation:



Having arrays defined in this way allows us still to treat the child edges of our tree as navigational axes: before we used a key such as `"age"` to traverse an edge, and now we use the number labelling the edge to traverse it and arrive at the child.

Formal definition. As our model is a tree, we will use tree domains as its base. A *tree domain* is a prefix-closed subset of $\mathbb{N}^*$. Without loss of generality we assume that for all tree domains D, if D contains a node $n \cdot i$, for $n \in \mathbb{N}^*$ then D contains all $n \cdot j$ with $0 \leqslant j < i$.

Let Σ be an alphabet. A **JSON tree** over Σ is a structure $J = (D, \mathit{Obj}, \mathit{Arr}, \mathit{Str}, \mathit{Int}, \mathcal{A}, \mathcal{O}, \mathit{val})$, where D is a tree domain that is partitioned by *Obj*, *Arr*, *Str* and *Int*, $\mathcal{O} \subseteq \mathit{Obj} \times \Sigma^* \times D$ is the object-child relation, $\mathcal{A} \subseteq \mathit{Arr} \times \mathbb{N} \times D$ is the array-child relation, $\mathit{val} : \mathit{Str} \cup \mathit{Int} \rightarrow \Sigma^* \cup \mathbb{N}$ is the string and number *value* function, and where the following holds:

1 For each node $n \in \mathit{Obj}$ and child $n \cdot i$ of n, $\mathcal{O}$ contains one triple $(n, w, n \cdot i)$, for a word $w \in \Sigma^*$.

2 The first two components of $\mathcal{O}$ form a *key*: if $(n, w, n \cdot i)$ and $(n, w, n \cdot j)$ are in $\mathcal{O}$, then $i = j$.

3 For each node $n \in \mathit{Arr}$ and child $n \cdot i$ of n, $\mathcal{A}$ contains the triple $(n, i, n \cdot i)$.

4 If n is in *Str* or *Int* then D cannot contain nodes of form $n \cdot u$.

5 The value function assigns to each string node in *Str* a value in Σ^* and to each number node in *Int* a natural number,

The usage of a tree domain is standard, and we have elected to explicitly partition the domain into four types of nodes: *Obj* for objects, *Arr* for arrays, *Str* for strings and *Int* for integers. The first and second conditions specify that edges between objects and their children are labelled with words, but we can only use each label one time per each node. The third condition specifies that the edges between arrays and their children are labelled with the number representing the order of children. The fourth condition simply states that strings and numbers must be leaves in our trees, and the fifth condition describes the value function *val*. Note that we have explicitly distinguished the four type of JSON documents (objects, arrays, strings and integers). This is important when modelling schema definitions for JSON, as we shall show later.

Throughout this paper we will use the term JSON tree and JSON interchangeably; that is, when we are given a JSON document we will assume that it is represented as a JSON tree. As already mentioned above, one important feature of our model is that when looking at any node of the tree, a subtree rooted at this node is again a valid JSON. We can therefore define, for a JSON tree J and a node n in J, a function $\mathit{json}(n)$ which returns the subtree of J rooted at n. Since this subtree is again a JSON tree, the value of $\mathit{json}(n)$ is always a valid JSON.

3.2 JSON and XML

Before continuing we give a few remarks about differences and similarities between JSON and XML, and how are these reflected in their underlying data models. We start by summarising the differences between the two formats.

1. *JSON mixes ordered and unordered data.* JSON objects are completely without order, but for arrays we can do random access depending on their position. On the other hand, XML enforces a strict order between the children of each node. Coding JSON as XML would imply permitting sibling traversal for some nodes, but disallowing it for others. We can do that with XML with some ad-hoc rules, but this is precisely what we do in our model in a much cleaner way.
2. *JSON Arrays are neither lists nor sets.* As we have explained, we have random access, but we do not have the possibility of sibling traversal. Enforcing this in languages such as XPath is a very cumbersome task.
3. *JSON trees are deterministic.* The property of JSON tree which imposes that all keys of each object have to be distinct makes JSON trees deterministic in the sense that if we have the key name, there can be at most one node reachable through an edge labelled with this key. On the other hand, XML trees are nondeterministic since there are no labels on the edges, and a node can have multiple children. As we will see, the deterministic nature of JSON can make some problems more difficult than in the XML setting.
4. *Value is not just in the node, but is the entire subtree rooted at that node.* Another fundamental difference is that in XML when we talk about values we normally refer to the value of an attribute in a node. On the contrary, it is common for systems using JSON to allow comparisons of the full subtree of a node with a nested JSON document, or even comparing two nodes themselves in terms of their subtrees. To be fair, in XML one could also argue this to be true, but unlike in the case of XML, these “structural” comparisons are intrinsic in most JSON languages, as we discuss in the following sections.

On the other hand, it is certainly possible to code JSON documents using the XML data format. In fact, the model of ordered unranked trees with labels and attributes, which serves as the base of XML, was shown to be powerful enough to code some very expressive database formats, such as relational and even graph data. However, both models have enough differences to justify a study of JSON on its own. This is particularly evident when considering navigation through JSON documents, where keys in each object have to be unique, thus allowing us to obtain values very efficiently. On the other hand, coding JSON as XML would require us to have keys as node labels, thus forcing a scan of all of the node's children in order to retrieve the value.

4. NAVIGATIONAL QUERIES OVER JSON

As JSON navigation instructions are too basic to serve as a complete query language, most systems have developed different ways of querying JSON documents. Unfortunately, there is no standard, nor general guidelines, about how documents are accessed. As a result the syntax and operations between systems vary so much that it would be almost impossible to compare them. Hence, it would be desirable to identify a common core of functionalities shared between these systems, or at least a general picture of how such query languages look like. Therefore we begin this section by reviewing the most common operations available in current JSON systems.

Here we mainly focus on the subdocument selecting functionalities of JSON query languages. By subdocument selecting we mean functionalities that are capable of finding or highlighting specific parts within JSON documents, either to be returned immediately or to be combined as new JSON documents. As our work is not intended to be a survey, we have not reviewed all possible systems available to date. However, we take inspiration from MongoDB's query language (which arguably has served as a basis for many other systems as well, see e.g. [2, 26, 31]), and JSONPath [14]and SQL++ [24], two other query languages that have been proposed by the community.

Based on this, we propose a navigational logic that can serve as a common core to define a standard way of querying JSON. We then define several extensions of this logic, such as allowing nondeterminism or recursion, and study how these affect basic reasoning task such as evaluation and satisfiability.

4.1 Accessing documents in JSON databases

Here we briefly describe how JSON systems query documents.

Query languages inspired by FLWR or relational expressions. There are several proposals to construct query languages that can merge, join and even produce new JSON documents. Most of them are inspired either by XQuery (such as JSONiq [29]) or SQL (such as SQL++ [24]). These languages have of course a lot of intricate features, and to the best of our knowledge have not been formally studied. However, in terms of JSON navigation they all seem to support basic JSON navigation instructions and not much more.

MongoDB's find function. The basic querying mechanism of MongoDB is given by the *find* function [22], therefore we focus on this aspect of the system[2]. The find function receives two parameters, which are both JSON documents. Given a collection of JSON documents and these parameters, the find function then produces an array of JSON documents.

The first parameter of the find function serves as a *filter*, and its goal is to select some of the JSON documents from the input. The second parameter is the *projection*, and as its name suggests, is used to specify which parts of the filtered documents are to be returned. Since our goal is specifying a navigational logic, we will only focus on the filter parameter, and on queries that only specify the filter. We return to the projection in Section 6. For more details we refer the reader to the current version of the documentation [22].

The basic building block of filters are what we call *navigation condition*, which can be visualised as expressions of the form $P \sim J$, where P is a JSON navigation instruction, $\sim$ is a comparison operator (MongoDB allows all the usual $<$, $\leqslant$, $=$, $\geqslant$, $>$, and several others operators) and J is a JSON document.

Example 1. *Assume that we are dealing with a collection of JSON files containing information about people and that we want to obtain the one describing a person named Sue. In MongoDB this can be achieved using the following query* `db.collection.find({name: {$eq: "Sue"}},{})`. *The initial part* `db.collection` *is a system path to find the collection of JSON documents we want to query. Next,* `"name"` *is a simple navigation instruction used to retrieve the value under the key* `"name"`. *Last, the expression* `{$eq: "Sue"}` *is used to state that the JSON document retrieved by the navigation instruction is equal to the JSON* `"Sue"`. *Since we are not dealing with projection, the second parameter is simply the empty document* `{}`. *Using the notation above we could also write this navigation condition as* $J[name] =$`"Sue"`.

[2]For a detailed study of other functionalities MongoDB offers see e.g. [8]. Note that this work does not consider the find function though.

Finally, navigation conditions can be combined using boolean operations with the standard meaning. Also note that filters always return entire documents. If we want a part of a JSON file we need to use projection.

Query languages inspired by XPath or relational expressions. The languages we analysed thus far offer very simple navigational features. However, people also recognized the need to allow more complex properties such as nondetermnistic navigation, expression filters and allowing arbitrary depth nesting through recursion. As a result, an adaptation of the XML query language XPath to the context of JSON, called JSONPath [14] was introduced and implemented (see e.g. https://github.com/jayway/JsonPath).

Based on these features, we first introduce a logic capturing basic queries provided by navigation instructions and conditions, and then extend it with non-determinism and recursion resulting in a logic resembling similar approaches over XML.

4.2 Deterministic JSON logic

The first logic we introduce is meant to capture JSON navigation instructions and other deterministic forms of querying such as MongoDB's find function. We call this logic *JSON navigation logic*, or JNL for short. We believe that this logic, although not very powerful, is interesting in its own right, as it leads to very lightweight algorithms and implementations, which is one of the aims of the JSON data format.

As often done in XML [12] and graph data [20], we define ours in terms of unary and binary formulas.

Definition 1 (JSON navigational logic). *Unary formulas φ, ψ and binary formulas α, β of the* JSON navigational logic *are expressions satisfying the grammar*

$$\begin{aligned} \alpha, \beta &:= \langle\varphi\rangle \mid X_w \mid X_i \mid \alpha \circ \beta \mid \varepsilon \\ \varphi, \psi &:= \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid [\alpha] \mid \\ &\qquad EQ(\alpha, A) \mid EQ(\alpha, \beta) \end{aligned}$$

where w is a word in Σ^*, i is a natural number and A is an arbitrary JSON document.

Intuitively, binary operators allow us to move through the document (they connect two nodes of a JSON tree), and unary formulas check whether a property is true at some point of our tree. For instance, X_w and X_i allow basic navigation by accessing the value of the key named w, or the ith element of an array respectively. They can subsequently be combined using composition or boolean operations to form more complex navigation expressions. Unary formulas serve as tests if some property holds at the part of the document we are currently reading. These also include the operator $[\alpha]$ allowing us to test if some binary condition is true starting at a current node (similarly, $\langle\varphi\rangle$ allows us to combine node tests with navigation). Finally, the comparison operators $EQ(\alpha, A)$ and $EQ(\alpha, \beta)$ simulate XPath style tests which check whether a current node can reach a node whose value is A, or if two paths can reach nodes with the same value. The difference with XML though, is that this value is again a JSON document and thus a subtree of the original tree.

The semantics of binary formulas is given by the relation $[\![\alpha]\!]_J$, for a binary formula α and a JSON J, and it selects pairs of nodes of J:

- $[\![\langle\varphi\rangle]\!]_J = \{(n, n) \mid n \in [\![\varphi]\!]_J\}$.
- $[\![X_w]\!]_J = \{(n, n') \mid (n, w, n') \in \mathcal{O}\}$.
- $[\![X_i]\!]_J = \{(n, n') \mid (n, i, n') \in \mathcal{A}\}$, for $i \in \mathbb{N}$.
- $[\![\alpha \circ \beta]\!]_J = [\![\alpha]\!]_J \circ [\![\beta]\!]_J$.
- $[\![\varepsilon]\!]_J = \{(n, n) \mid n \text{ is a node in } J\}$.

For the semantic of the unary operators, let us assume that D is the domain of J.

- $[\![\top]\!]_J = D$.
- $[\![\neg\varphi]\!]_J = D - [\![\varphi]\!]_J$.
- $[\![\varphi \wedge \psi]\!]_J = [\![\varphi]\!]_J \cap [\![\psi]\!]_J$.
- $[\![\varphi \vee \psi]\!]_J = [\![\varphi]\!]_J \cup [\![\psi]\!]_J$.
- $[\![[\alpha]]\!]_J = \{n \mid n \in D$ and there is a node n' in D such that $(n, n') \in [\![\alpha]\!]_J\}$
- $[\![EQ(\alpha, A)]\!]_J = \{n \mid n \in D$ and there is a node n_1 in D such that $(n, n_1) \in [\![\alpha]\!]_J$ and $json(n_1) = A\}$
- $[\![EQ(\alpha, \beta)]\!]_J = \{n \mid n \in D$ and there are nodes n_1, n_2 in D such that $(n, n_1) \in [\![\alpha]\!]_J$, $(n, n_2) \in [\![\beta]\!]_J$, and $json(n_1) = json(n_2)\}$.

Typically, most systems allow jumping to the last element of an array, or the j-th element counting from the last to the first. To simulate this we can allow binary expressions of the form X_i, for an integer $i < 0$, where -1 states the last position of the array, and $-j$ states the j-th position starting from the last to the first. Having this dual operator would not change any of our results, but we prefer to leave it out for the sake of readability.

Algorithmic properties of JNL. As promised, here we show that JNL is a logic particularly well behaved for database applications. For this we study the evaluation problem and satisfiability problem associated with JNL. The EVALUATION problem asks, on input a JSON J, a JNL unary expression φ and a node n of J, whether n is in $[\![\varphi]\!]_J$. The SATISFIABILITY problem asks, on input a JNL expression φ , whether there exists a JSON J such that $[\![\varphi]\!]_J$ is nonempty. We start with evaluation, showing that JNL indeed matches the "lightweight" spirit of the JSON format and can be evaluated very efficiently:

PROPOSITION 1. *The* EVALUATION *problem for JNL can be solved in time* $O(|J| \cdot |\varphi|)$.

For this result, we can reuse techniques for XPath evaluation (see e.g. [27, 15]). However, the presence of the $EQ(\alpha, \beta)$ operator forces us to refine these techniques in a non-trivial way. A straightforward way of incorporating this predicate into XPath algorithms is to pre-process all pairs of nodes to see which pairs have equal subtrees, but this only gives us a quadratic algorithm. Instead, we transform our JNL formula into an equivalent non recursive monadic datalog program with stratified negation [16], and show how to evaluate the latter by doing equality comparisons "online" as they appear.

Next, we move to satisfiability, showing that the complexity of the problem is optimal, considering that JNL can emulate propositional formulas.

PROPOSITION 2. *The* SATISFIABILITY *problem for JNL is* NP*-complete. It is* NP*-hard even for formulas neither using negation nor the equality operator.*

It might be somewhat surprising that the positive fragment without data comparisons is not trivially satisfiable. This holds due to the fact that each key in an object is unique, so a formula of the form $[X_a \circ \langle[X_1]\rangle] \wedge [X_a \circ \langle[X_b]\rangle]$ is unsatisfiable because it forces the value of the key a to be both an array and a string at the same time.

4.3 Extensions

Although the base proposal for JNL captures the deterministic spirit of JSON, it is somewhat limited in expressive power. Here we propose two natural extensions: the ability to non-deterministically select which child of a node is selected, and the ability to traverse paths of arbitrary length.

Non-determinism. The path operators X_w and X_i can be easily extended such that they return more than a single child; namely, we can permit matching of regular expressions and intervals, instead of simple words and array positions.

Formally, non-deterministic JSON logic extends binary formulas of JNL by the following grammar:

$$\alpha, \beta \quad := \quad \langle\varphi\rangle \mid X_e \mid X_{i:j} \mid \alpha \circ \beta \mid \varepsilon$$

where e is a subset of Σ^* (given as a regular expression), and $i \leqslant j$ are natural numbers, or $j = +\infty$ (signifying that we want any element of the array following i). The semantics of the new path operators is as follows:

- $[\![X_e]\!]_J = \{(n, n') \mid$ there is $w \in L(e)$ such that $(n, w, n') \in \mathcal{O}\}$.
- $[\![X_{i:j}]\!]_J = \{(n, n') \mid$ there is $i \leqslant p \leqslant j$ such that the triple (n, p, n') is in $\mathcal{A}\}$.

Recursion. In order to allow exploring paths of arbitrary length we add the Kleene star to our logic. That is, recursive JNL allows $(\alpha)^*$ as a binary formula (as usual we normally omit the brackets when the precedence of operators is clear). The semantics of $(\alpha)^*$ is given by

$$[\![(\alpha)^*]\!] = [\![\varepsilon]\!]_J \cup [\![\alpha]\!]_J \cup [\![\alpha \circ \alpha]\!] \cup [\![\alpha \circ \alpha \circ \alpha]\!]_J \cup \ldots.$$

So what happens to the evaluation and satisfiability when we extend this logic? For the case of evaluation, we can easily show that the linear algorithm is retained as long as we do not have the binary equality operator $EQ(\alpha, \beta)$. Indeed, in this case, the evaluation can be done using the classical PDL model checking algorithm [1, 10] with small extensions which account for the specifics of the JSON format. However, we are not able to extend the linear algorithm for the full case, because an expression of the form $EQ(\alpha, \beta)$ might require checking all pairs of nodes in our tree for equality, resulting in a jump in complexity.

PROPOSITION 3. *The evaluation problem for JNL with non-determinism and recursion can be solved in time $O(|J|^3 \cdot |\varphi|)$, and in time $O(|J| \cdot |\varphi|)$ if the formula does not use the predicate $EQ(\alpha, \beta)$.*

For satisfiability the situation is radically different, as the combination of recursion, non-determinism and the binary equalities ends up being too difficult to handle.

PROPOSITION 4. *The* SATISFIABILITY *problem is undecidable for non-deterministic recursive JNL formulas, even if they do not use negation.*

However, if we rule out the equality operator we can show much better bounds. For the full non-deterministic, recursive JNL (without equalities) the satisfiability problem is the same as other similar fragments such as PDL. For (non-recursive) non-deterministic JNL the problem is slightly easier.

PROPOSITION 5. *The* SATISFIABILITY *problem is:*

- PSPACE-*complete for non-deterministic, non-recursive JNL without the $EQ(\alpha, \beta)$ operator.*
- EXPTIME-*complete for non-deterministic, recursive JNL without the $EQ(\alpha, \beta)$ operator.*

Note that PSPACE-hardness for satisfiability follows easily from the fact that we now allow regular expressions in our edges: Given a regular expression e, we have that the e is universal if and only if the query $[X_{\Sigma^*}] \wedge \neg[X_e]$ is not satisfiable. However, in the proof of this proposition we in fact show that the problem remains PSPACE-hard even when the only regular expression which is not a word in a X_e axis is Σ^*. One can also show that PSPACE-hardness remains when one only considers JSON documents without object values.

5. SCHEMA DEFINITIONS FOR JSON

Having dealt with navigational primitives for querying JSON, our next task is to analyse JSON Schema definitions. We focus solely on the JSON Schema specification [17], which is, up to our best knowledge, the only attempt to define a general schema language for JSON documents. The JSON Schema specification is currently in its fourth draft, and on its way of becoming an IETF standard.

5.1 JSON Schema

As before, we first briefly present how JSON Schema works. We remark again that our intention is not to provide a full analysis for the specification, but rather show how the navigation works, with the aim of obtaining a logic that can capture JSON Schema. We thus concentrate on a core fragment that is equivalent to the full specification; we refer to [28] for more details and a full formalisation of this core.

Every JSON schema is JSON document itself. JSON Schema can specify that a document must be any of the different types of values (objects, arrays, strings or numbers); and for each of these types there are several keywords that help shaping and restricting the set of documents that a schema specifies. The most important keyword is the "type" keyword, as it determines the type of a value that has to be validated against the schema: a document of the form {`"type":"string"`, ...} specifies string values, {`"type":"number"`, ...} specifies number values, {`"type":"object"`, ...} specifies objects and {`"type":"array"`, ...} specifies arrays. In addition to the type keyword, each schema includes a number of other pairs that shape the documents they describe.

We now describe each of the four types of basic schemas. Table 1 contains a list of all keywords available for each of these schemas.

String schemas. String schemas are those featuring the `"type":"string"` pair. Additionally, they may include the pair `"pattern"`:"*regexp*", for *regexp* a regular expression over Σ, which validates only against those strings that belong to the language of this expression. For example, {`"type":"string"`} and {`"type":"string"`,`"pattern"`:"$(01)^+$"} are string schemas. The first schema validates against any string, and the second only against strings built from 0 or 1.

Number schemas. For numbers, we can use the pair `"minimum"`:i to specify that the number is at least i, `"maximum"`:i to specify that the number is at most i, and `"multipleOf"`:i to specify that a number must be a multiple of i. Thus for example {`"type":"number"`,`"maximum"`:12,`"multipleOf"`:4} describes numbers 0, 4, 8 and 12.

Object schemas. Besides the `"type":"object"` pair, object schemas may additionally have the following:

- Pairs `"minProperties":i` and `"maxProperties":j`, to specify that an object has to have at least i and/or at most j key-value pairs.

- a pair `"required"`:$[k_1, \ldots, k_n]$, where each k_i is a string value. This keyword mandates that the specified object values must contain pairs with keys $k_1, \ldots, k_n$.

- a pair `"properties"`:$\{k_1 : J_1, \ldots, k_m : J_m\}$, where each k_i is a string value and each J_i is itself a JSON Schema. This keyword states that the value of each pair with key k_i must validate against schema J_i.
- A pair `"patternProperties"`:{"e_1":$J_1, \ldots,$ "e_ℓ":J_ℓ}, where each e_i is a regular expression over Σ and each J_i is a JSON schema. This keyword works just like `properties`, but now any value under any key that conforms to the expression exp_i must satisfy J_i.

- finally, the pair `"additionalProperties"`:J, where J is a JSON schema. This keyword presents a schema that must be satisfied by all values whose keys do not appear neither in `properties` nor conform to the language of an expression in `patternProperties`. For example, the schema in the Figure 2 below specifies objects where the value under "name" must be a string, the value under any key of the form `a(b|c)a` must be an even number, and the value under any key which is neither "name" nor conforms to the expression above must always be the number 1.

Keywords for string schemas:

- "type":"string"   - "pattern": exp

Keywords for number schemas:

- "type":"number"   -"multipleOf": i
- "minimum": i - "maximum": i

Keywords for array schemas:

- "items":$[J_1, \dots, J_n]$
- "uniqueItems":true
- "additionalItems":J

Keywords for object schemas:

- "type":"object"   - "required": $[\ k_1, \dots, k_n]$
- "minProperties": i   - "maxProperties": i
- "properties":$\{k_1 : J_1, \dots, k_m : J_m\}$
- "patternProperties":$\{$"e_1":$J_1, \dots,$ "e_ℓ":$J_\ell\}$
- "additionalProperties": J

Boolean combination and comparisons:

- "anyOf": $[\ J_1, \dots, J_n]$ - "allOf": $[\ J_1, \dots, J_m]$
- "not": J - "enum": $[\ A_1, \dots, A_n]$

Table 1: The form for all keyords in JSON schema. Here i is always a natural number, J and each J_i are JSON schemas, $A_1, \dots, A_n$ are JSON documents, each k_i is a string value (k stand for key); and exp and each exp_i are regular expressions over the alphabet Σ of strings.

```
{
   "type": "object",
   "properties": {
      "name": {"type":"string"},
      },
   "patternProperties: {
      "a(b|c)a": {"type":"number", "multipleOf": 2}
      },
   "additionalProperties: {
      "type": "number",
      "minimum": 1
      "maximum": 1
   }
}
```

Figure 2: Example of an object schema.

Array schemas. Array schemas are specified with the "type": "array" keyword. For arrays there are two ways of specifying what kind of documents we find in arrays. We can use a pair "items":$[J_1, \dots, J_n]$ to specify a document with an array of n elements, where each i-th element must satisfy schema J_i. We can also use "additionalItems":J to specify that all elements in the array must satisfy schema J. If both keywords are used together, then we allow the array to have more values than those specified with items, as long as they agree with the schema specified in additionalItems. Finally, one can include the pair "uniqueItems":true to force arrays whose elements are all distinct from each other. For example, the following schema validates against arrays of at least 2 elements, where the first two are strings and the remaining ones, if they exists, are numbers.

```
{
   "type": "array",
   "items": [{"type":"string"}, {"type":"string"}],
   "additionalItems": {"type":"number"},
   "uniqueItems":true
}
```

Boolean combinations. The last feature in JSON Schema are boolean combinations. These allow us to specify that a document must validate against two schemas, against at least one schema, or that it must not validate against a schema. For example, the schema "not":{"type":"number","multipleOf":2} validates against any odd number, or any document which is not a number.

5.2 JSON Schema Logic

In order to capture the JSON Schema specification with a logical formalism, we isolate navigation and atomic tests into two different sets of operators. Let us start with atomic operations, which are basically a rewriting of most JSON Schema keywords into our framework. We allocate them in the set *NodeTests.*

Formally, *NodeTests* contains the predicates *Arr*, *Obj*, *Str*, *Int* and *Unique*, plus a predicate $Pattern(e)$ for each regular expression e built from Σ, predicates $Min(i)$ and $Max(i)$ for each integer i, a predicate $MultOf(i)$ for each $i \geqslant 0$, predicates $MinCh(k)$ and $MaxCh(k)$ for each $k \geqslant 0$ and a predicate $\sim(A)$ for each JSON document A. The semantics of these predicates is given by the relation $\models$, that states whether an atomic predicate holds for a given node n of a JSON J.

- $(J, n) \models Arr$ iff. $n \in Arr$.   - $(J, n) \models Obj$ iff. $n \in Obj$.
- $(J, n) \models Str$ iff. $n \in Str$.   - $(J, n) \models Int$ iff. $n \in Int$.
- $(J, n) \models Pattern(e)$ iff. $val(n)$ is a string in $L(e)$.
- $(J, n) \models Min(i)$ iff. $val(n)$ is a number greater than i.
- $(J, n) \models Max(i)$ iff. $val(n)$ is a number smaller than i.
- $(J, n) \models MultOf(i)$ iff. $val(n)$ is a multiple of i.
- $(J, n) \models MinCh(i)$ iff. n has at least i children.
- $(J, n) \models MaxCh(i)$ iff. n has at most i children.
- $(J, n) \models Unique$ iff. $n \in Arr$ and all of its children are different JSON values: for each $n' \neq n''$ such that (n, p, n') and (n, q, n'') belong to $\mathcal{A}$ we have that $json(n') \neq json(n'')$.
- $(J, n) \models \sim(A)$ iff. $json(n) = A$.

With *NodeTests* we cover all atomic features of JSON Schema. All that remains is the navigation, which in JSON Schema is given by the keywords properties, patternProperties, additionalProperties and required, for objects, and items and additionalItems for arrays. These forms of navigation suggest using existential and universal modalities. For instance, the keyword patternProperties specifies a schema that must be validated by *all* values whose keys satisfy a regular expression, and "required": $[w]$ demands that there must *exist* a children with key w. Thus, to fully define our logic, we augment our node tests with universal and existential modalities, as well as boolean combinations.

Definition 2. *Formulas in the* JSON schema logic *(JSL) are expressions satisfying the grammar*

$$\begin{aligned}\varphi, \psi \quad := \quad & \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \psi \in \mathit{NodeTests} \mid \\ & \Box_e\varphi \mid \Box_{i:j}\varphi \mid \Diamond_e\varphi \mid \Diamond_{i:j}\varphi\end{aligned}$$

where e is a subset of Σ^* (given as a regular expression), $i \leqslant j$ are natural numbers, or $j = +\infty$ (signifying that we want any element of the array following i).

As with JSON navigational logic, we can also obtain a deterministic version of JSL by restricting the syntax to use only modal operators $\Box_w$ and $\Box_i$, and $\Diamond_w$ and $\Diamond_i$; for a word $w \in \Sigma^*$ and a natural number i.

The semantics is given by extending the relation $\models$.

- $(J, n) \models \top$ for every node n in J.

- $(J, n) \models \neg\varphi$ iff. $(J, n) \not\models \varphi$.

- $(J, n) \models \varphi \wedge \psi$ iff. $(J, n) \models \varphi$ and $(J, n) \models \psi$.

- $(J, n) \models \varphi \vee \psi$ iff. either $(J, n) \models \varphi$ or $(J, n) \models \psi$.

- $(J, n) \models \Diamond_e\, \varphi$ iff. there is a word $w \in L(e)$ and a node n' in J such that $(n, w, n') \in \mathcal{O}$ and $(J, n') \models \varphi$

- $(J, n) \models \Diamond_{i:j}\, \varphi$ iff. there is $i \leqslant p \leqslant j$ and a node n' in J such that $(n, p, n') \in \mathcal{A}$ and $(J, n') \models \varphi$

- $(J, n) \models \Box_e\, \varphi$ iff. $(J, n') \models \varphi$ for all nodes n' such that $(n, w, n') \in \mathcal{O}$ for some $w \in L(e)$.

- $(J, n) \models \Box_{i:j}\, \varphi$ iff. $(J, n') \models \varphi$ for all nodes n' such that $(n, p, n') \in \mathcal{A}$ for some $i \leqslant p \leqslant j$.

In order to present our results regarding JSL and JSON Schema, we abuse notation and write $J \models \psi$ whenever $(J, r) \models \psi$, where r is the root of J.

Expressive power. As promised we show that JSL can capture JSON schema. In order to present this result we informally speak of the validation relation of JSON Schema, and say that a JSON S validates against J whenever J is in accordance to all keywords present in S. We refer to [28] for more details on the semantics. As usual, we say that JSON Schema and JSL are equivalent in expressive power if for any JSON Schema S there exists a *JSL* formula ψ_S such that for every JSON document J we have that J validates against S if and only if $J \models \psi_S$; and conversely, for any JSL formula φ there exists a JSON Schema S_φ such that for every JSON document J we have that $J \models \varphi$ if and only if J validates against S_φ.

Theorem 1. *JSL and JSON Schema are equivalent in expressive power.*

Comparing JSL and JNL. Next, we consider how the navigation logic of Section 4 compares to the schema logic JSL. Even though the starting point of the two logics is different, we next show that the two logics are essentially the same, their expressivity differing simply because of the different atomic predicates. More precisely, we have:

Theorem 2. *Non-deterministic JNL not using the equality $EQ(\alpha, \beta)$ and non-deterministic JSL using only the node test $\sim (A)$ are equivalent in terms of expressive power. More precisely:*

- *For every formula φ^S in JSL there exists a unary formula φ^N in JNL such that for every JSON J:*
$$[\![\varphi^N]\!]_J = \{n \in J \mid (J, n) \models \varphi^S\}.$$

- *For every unary formula φ^N in JNL there exists a φ^S in JSL such that for every JSON J:*
$$[\![\varphi^N]\!]_J = \{n \in J \mid (J, n) \models \varphi^S\}.$$

In the proof above, we also show that going from JSL to JNL takes only polynomial time, while the transition in the other direction can be exponential. This implies that the upper bounds for JNL are valid for JSL, while the lower bounds transfer in the opposite direction (without taking into account node tests).

Algorithmic Properties. Since JSL is designed to be a schema logic to validate trees, we specify a boolean Evaluation problem: the input is a JSON J and a JSL expression φ, and we decide whether $J \models \varphi$.

Proposition 6. *The* Evaluation *problem for JSL can be solved in time $O(|J|^2 \cdot |\varphi|)$, and in $O(|J| \cdot |\varphi|)$ when φ does not use the Unique predicate.*

From Theorem 1 we obtain as a corollary that the validation problem for JSON Schema has the same bounds. This was already shown in [28]. Next, we study the Satisfiability problem, which receives a formula φ as input and consists of deciding whether there is any JSON tree J such that $J \models \varphi$. Here we need to be careful with the encoding we choose, as the interplay between *Unique* and $\Diamond_i\top$ immediately forces a node with exponentially many different children when i is given in binary. In terms of results, this means that our algorithms raise by one exponential, although we do not know if this increase is actually unavoidable.

Proposition 7. *The* Satisfiability *problem for JSL is in* Expspace*, and* Pspace*-complete for expressions without Unique.*

We remark that the Satisfiability problem is important in the context of JSON Schema. For example, the community has repeatedly stated the need for algorithms that can learn JSON Schemas from examples. We believe that understanding basic tasks such as satisfiability are the first steps to proceed in this direction.

5.3 Adding recursion

The JSON Schema specification also allows defining statements that will be reused later on. We have so far ignored this functionality, and to capture it we will need to define the same operator in our logic. As we will see, this lifts the expressive power of JSL away from even the recursive version of our navigational logic; and is very similar to certain forms of tree automata.

Let us explain first how recursion is added into JSON Schema. The idea is to allow to an additional keyword, of the form `{$ref: <path>}`, where `<path>` is a navigation instruction. This instruction is used within the same document to fetch other schemas that have been predefined in a reserved `definitions` section[3]. For example, the following schema validates against any JSON which is not a string following the specified pattern.

[3]Definitions and references can also be used to fetch schemas in different documents or even domains; here we just focus on the recursive functionality.

```
{
    "definitions": {
        "email": {
            "type": "string",
            "pattern": "[A-z]*@ciws.cl"
        }
    },
    "not": {"$ref": "#/definitions/email"}
}
```

As we have mentioned, different schemas are defined under the `definitions` section of the JSON document, and these definitions can be reused using the `$ref` keyword. In the example above, we use `{"$ref": "#/definitions/email"}` to retrieve the schema `{"type": "string", "pattern": "[A-z]*@ciws.cl"}`. These definitions can be nested within each other, but semantics is currently defined only for a fragment with limited recursion. We come back to this issue after we define a logic capturing these schemas.

Recursive JSL. The idea of this logic is to capture the recursive functionalities present in JSON Schema: there is a special section where one can define new formulas, which can be then re-used in other formulas.

Fix an infinite set $\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \dots\}$ of symbols. A *recursive* JSL formula is an expression of the form

$$\begin{array}{rcl} \gamma_1 & = & \varphi_1 \\ \gamma_2 & = & \varphi_2 \\ & \vdots & \\ \gamma_m & = & \varphi_m \\ \psi & & \end{array} \qquad (1)$$

where each γ_i is a symbol in Γ and φ_i, ψ are JSL formulas over the extended syntax that allows $\gamma_1, \dots, \gamma_m$ as atomic predicates. Here each equality $\gamma_i = \varphi_i$ is called a definition, and ψ is called the base expression.

The intuition is that each γ_i is one of the references of JSON Schema. Before moving to the semantics, let us show the way recursive JSL formulas work.

Example 2. *Consider the expression Δ given by*

$$\begin{array}{l} \gamma_1 = \Box_{\Sigma^*}\gamma_2 \\ \gamma_2 = (\Diamond_{\Sigma^*}\top) \wedge (\Box_{\Sigma^*}\gamma_1) \\ \gamma_1 \end{array}$$

Intuitively, γ_1 holds in a node n if n either has no children, or if γ_2 holds all of its children. On the other hand, γ_2 holds in a node n if this node has at least one child, and γ_1 holds in all children of n. Finally, the base expression simply states that γ_1 has to hold in the root of the document (recall that a schema statement is evaluated at the root). The intuition for Δ is, then, that it should hold on every tree such that each path from the root to the leaves is of even length.

Well-formed recursive JSL. As usual in formalisms that mix recursion and negation, giving a formal semantics for JSON Schema is not a straightforward task. As an example of the problems we face, consider the following JSL expression.

$$\begin{array}{l} \gamma_1 = \neg\gamma_1 \\ \gamma_1 \end{array}$$

Of course, we can always give a logical interpretation to this formula, but we argue that this expression does not specify any real restriction on JSON documents, and thus any semantics we establish will not be intuitive from the point of view of defining schemas for JSON.

The most straightforward way to avoid these issues is by imposing a strict acyclicity condition on definitions. However, we can in fact work with a much milder restriction that we call *well-formedness*. This restriction was introduced in [28] for the case of JSON Schema, but we can seemingly define well-formed recursive JSL expressions, which can then be shown to capture well-formed recursive JSON Schemas.

For a recursive JSL expression Δ of the form (1) defined above, we define the precedence graph of Δ as a graph whose nodes are $\gamma_1, \dots, \gamma_m$ and where there is an edge from γ_i to γ_j if γ_j appears in the expression φ_i of the definition $\gamma_i = \varphi_i$, but only if this appearance is *not under the scope of a modal operator*. We then say that Δ is *well-formed* if its precedence graph is acyclic.

Example 3. *The definition $\gamma_1 = \neg\gamma_1$ clearly creates a cyclic precedence graph, as the construction mandates a self-loop in the node corresponding to γ_1. On the other hand, the recursive JSL expression in Example 2 is indeed well-formed. It does introduce cycles in the definitions, but no edges are added into the precedence graph of such expression because formula symbols are always under the scope of a modal operator.*

Semantics. How do we then define the semantics of well-formed expressions? If Δ is a recursive JSL expression that is completely acyclic, then we can simply replace the symbols in Γ by their respective definitions. But we cannot do this with every well-formed expression, because some of them can have cycles in the definitions (under a scope of a modal operator), and thus we would never stop replacing symbols. This is the case, for instance, in Example 2.

However, the key thing to notice is that we only need to do this as many times as the height of the JSON tree we are trying to validate. More precisely, let J be a JSON tree of height h and Δ a well-formed recursive JSL expression of the form (1). Construct a (non-recursive) JSL expression $\mathit{unfold}_J(\psi)$ by replacing each symbol γ_i in ψ by the corresponding definition φ_i, but stop once every symbol from Γ is under the scope of at least $h + 1$ modal operators. Afterwards, replace all the remaining symbols γ_i for the symbol $\bot$ (shorthand for $\neg\top$). If Δ is well-formed, then this procedure is guaranteed to stop, because every time we come back to the same symbol γ when replacing, we know that this symbol has to be under the scope of at least one more modal operator. Then, we define the satisfaction relation based on the satisfaction of the constructed formula ψ, so that $J \models \Delta$ if and only if $J \models \mathit{unfold}_J(\psi)$.

Example 4. *Suppose that we need to evaluate the expression in Example 2 over the tree J of height 4. Then we would only need to keep rewriting ψ until all of γ_1 and γ_2 are under at least 5 modal operators. The query obtained corresponds to*

$$\Box_{\Sigma^*}(\Diamond_{\Sigma^*}\top \wedge \Box_{\Sigma^*}\Box_{\Sigma^*}(\Diamond_{\Sigma^*}\top \wedge \Box_{\Sigma^*}\Box_{\Sigma^*}\gamma_2))$$

To finalise we create the formula $\mathit{unfold}_J(\gamma_1)$ by replacing the symbol γ_2 in the expression above for $\bot$. Now the evaluation of the original expression over J corresponds to the evaluation of $\mathit{unfold}_J(\gamma_1)$ over J.

Expressive Power. As promised, one can show that recursive JSL captures the recursive definition of JSON Schema. We use the same notation as for Theorem 1.

THEOREM 3. *Well-formed recursive JSL and well-formed recursive JSON Schema are equivalent in expressive power.*

Comparing JSL to navigational logic JNL, we can again show that different atomic predicates force them to have different expressive power: for instance JSL can not express the $EQ(\alpha, \beta)$ operator, while JNL can not cover unique items. On the other hand, if $EQ(\alpha, \beta)$ is not allowed, then we conjecture recursive non-deterministic JNL to be strictly less expressive than recursive JSL without unique items, due to the more powerful form of recursion available in the language.

We finish with a few remarks on the expressive power of recursive JSL expressions. First, let us note that, even if one would like to compare recursive JSL or JSON Schema against XML Schema definitions such as DTDs or XML Schema, or even to Monadic Second Order (MSO), we cannot do it in a direct way since the models of XML and JSON have several important differences, and it is not immediate to express JSON as a relational structure. However, just for the sake of establishing a comparison we can assume that the set Σ^* of possible keys is fixed and finite, and that we do not consider arrays. We can then just focus on a relational representation of JSON that has one binary relation $\mathcal{O}_w$ for each word w in our set of keys, plus all predicates specified in *NodeTests*. We can then show the following (as usual MSO is said be equivalent to JSL if for every JSL expression we can create a boolean MSO formula accepting the same trees, and vice-versa):

PROPOSITION 8. *Well-formed recursive JSL is equivalent to MSO, if the set Σ^* of possible keys is fixed and finite, and if we do not consider arrays.*

What about arrays? The first observation is that the presence of arrays introduces the atomic test *Unique*, which can express properties not definable in MSO:

EXAMPLE 5. *The following recursive JSL expression accepts only JSON documents representing complete binary trees:*

$$\begin{aligned} \gamma = \neg(\Diamond_1\top) \vee (&\mathit{MinCh}(2) \wedge \mathit{MaxCh}(2) \wedge \\ &\neg\,\mathit{Unique} \wedge \Box_{1:2}\gamma) \\ \gamma \quad & \end{aligned}$$

Every node is an array with either no child or with two children both satisfying γ, and the $\neg$ Unique restriction forces the two children of each node to be equal.

Even if we rule out *Unique*, it is still not easy to exactly pinpoint what JSL can do. On one hand, JSON arrays are ordered, in the sense that the first item can be distinguished from the second. But on the other hand the reasoning between elements in arrays in JSON is very limited. For example, we cannot use JSL to specify that after an element satisfying a formula φ we must have an element satisfying another formula ψ.

Evaluation. The first thing to note is that the semantics for recursive JSL, as defined previously, leads to very inefficient evaluation algorithms: the rewriting $\mathit{unfold}_J(\psi)$ may well be of exponential size with respect to the original query, even if J contains a single node. However, we can show that evaluating recursive JSL expressions remains in PTIME in combined complexity, although the succinctness introduced by the possibility of reusing definitions makes the problem PTIME-hard. Our algorithm consists on evaluating all subtrees of J in a bottom-up fashion, proceeding to higher height levels of J only when all the previous levels have already been computed. The algorithm resembles the evaluation of Datalog programs with stratified negation.

PROPOSITION 9. *The* EVALUATION *problem for recursive JSL expressions over JSON trees is* PTIME*-complete in combined complexity.*

Satisfiability. The most common way of building a satisfiability algorithm in schema formalisms for trees is to show that they are equivalent to some class of tree automata whose non-emptiness problem can be shown to be decidable. We use the same ideas, albeit we need to introduce a specific model of automata that can capture our formalism. Interestingly, we show that the *Unique* predicate can also be handled in this case, albeit with an exponential blowup. To show this we encode *Unique* as a special local constraint, as done in e.g. [4, 19]. Again, we do not know if this blowup is unavoidable.

PROPOSITION 10. *The* SATISFIABILITY *problem for recursive JSL expressions is in* 2EXPTIME*, and* EXPTIME*-complete for expressions without the Unique predicate.*

6. FUTURE PERSPECTIVES

In this work we present a first attempt to formally study the JSON data format. To this end, we describe the underlying data model for JSON, and introduce logical formalisms which capture the way JSON data is accessed and controlled in practice. Through our results we emphasise how the new features present in JSON, affect classical results known from the XML context, and highlight that there is a need for developing new techniques to tackle main static tasks for JSON languages. And while some of these features have been consider in the past (e.g. comparing subtrees [5, 33], or an infinite number of keys [6]), it is still not entirely clear how these properties mix with the deterministic structure of JSON trees, thus providing an interesting ground for future work.

Apart from these fundamental problems that need to be tackled, there is also a series of practical aspects of JSON that we did not consider. In particular, we identify three areas that we believe warrant further study, and where the formal framework we propose could be of use in understanding the underlying problems.

MongoDB's projection. While we have presented a navigational logic that can capture the way MongoDB filters JSON documents within its find function, we have left out the second argument of the find function, known as projection. In its essence, the idea of the projection argument is to select only those subtrees of input documents that can be reached by certain navigation instructions, thus defining a JSON to JSON transformation. Although well-defined, the projection in MongoDB is quite limited in expressive power, and does not allow a lot of interaction between filtering and projecting. We believe that this is an interesting ground for future work, as there are many fundamental questions regarding the expressive power of these transformations, and their possible interactions with schema definitions.

Streaming. Another important line of future work is streaming. Indeed, the widespread use of JSON as a means of communicating information through the Web demands the usage of streaming techniques to query JSON documents or validate document against schemas. Streaming applications most surely will be related to APIs, either to be able to query data fetched from an API without resorting to store the data (for example if we are in a mobile environment) or to validate JSONs on-the-fly before they are received by APIs. In contrast with XML (see, e.g., [30], we suspect that our deterministic versions of both JNL and JSL might actually be shown to be evaluated in a streaming context with constant memory requirements when tree equality is excluded from the language.

Documenting APIs. One of the main uses of JSON Schema is the *Open API initiative* [25], an endeavour founded in early 2016 that intends to build an open documentation of RESTful APIs available worldwide. This initiative uses JSON Schema to specify the inputs and the outputs of a large number of APIs, and is trying to describe which parts of the API output actually come from the input. A formal perspective in these tasks will certainly be well appreciated by the community. Furthermore, there is also the problem of documenting how JSON APIs interact with real databases. For example, when large organisations expose data they sometimes use APIs instead of proper database views, even if it is for internal uses. This immediately raises the problem of exchanging data that may now not reside on real databases, but is exposed only by means of JSON. We believe these problems raise interesting questions for future work.

Acknowledgements

We would like to thank the anonymous reviewers for many helpful comments. Reutter, Suárez and Vrgoč were funded by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004. Bourhis and Vrgoč were partially funded by the STIC AMSUD project Foundations of Graph Structured Data (Fog). Bourhis was partially funded by the DeLTA project (ANR-16-CE40-0007).

7. REFERENCES

[1] N. Alechina and N. Immerman. Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the IGPL*, 8(3):325–337, 2000.

[2] AranbgoDB Inc. The ArangoDB database. https://www.arangodb.com/, 2016.

[3] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. Relative expressiveness of nested regular expressions. In *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012*, pages 180–195, 2012.

[4] Bruno Bogaert and Sophie Tison. Equality and disequality constraints on direct subterms in tree automata. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 159–171. Springer, 1992.

[5] Bruno Bogaert and Sophie Tison. Equality and disequality constraints on direct subterms in tree automata. In *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, pages 161–171, 1992.

[6] Adrien Boiret, Vincent Hugot, Joachim Niehren, and Ralf Treinen. Deterministic automata for unordered trees. In *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014.*, pages 189–202, 2014.

[7] Mikoaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and xml reasoning. *Journal of the ACM (JACM)*, 56(3):13, 2009.

[8] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. A formal presentation of mongodb (extended version). *CoRR*, abs/1603.09291, 2016.

[9] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. 2014.

[10] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, 1993.

[11] ECMA. The JSON Data Interchange Format. http://www.ecma-international.org/publications/standards/Ecma-404.htm, 2013.

[12] Diego Figueira. *Reasoning on words and trees with data. (Raisonnement sur mots et arbres avec données)*. PhD thesis, École normale supérieure de Cachan, France, 2010.

[13] Python Software Foundation. Python programming language. https://www.python.org/, 2016.

[14] Stefan Gössner and Stephen Frank. JSONPath. http://goessner.net/articles/JsonPath/, 2007.

[15] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[16] Georg Gottlob, Christoph Koch, and Klaus U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, March 2006.

[17] Internet Engineering Task Force (IETF). The JavaScript Object Notation (JSON) Data Interchange Format. https://tools.ietf.org/html/rfc7159, March 2014.

[18] json-schema.org: The home of json schema. http://json-schema.org/.

[19] Wong Karianto and Christof Löding. Unranked tree automata with sibling equalities and disequalities. In *International Colloquium on Automata, Languages, and Programming*, pages 875–887. Springer, 2007.

[20] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *J. ACM*, 63(2):14, 2016.

[21] Zhen Hua Liu, Beda Christoph Hammerschmidt, and Doug McMahon. JSON data management: supporting schema-less development in RDBMS. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1247–1258, 2014.

[22] MongoDB Inc. The MongoDB3.0 Manual. https://docs.mongodb.org/manual/, 2015.

[23] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. *Caine*, 2009:157–162, 2009.

[24] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. *CoRR*, abs/1405.3631, 2014.

[25] The Open API Initiative. https://openapis.org/, 2016.

[26] OrientDB LTD. The OrientDB database. http://orientdb.com/, 2016.

[27] Pawel Parys. Xpath evaluation in linear time with polynomial combined complexity. In *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, pages 55–64, 2009.

[28] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 263–273, 2016.

[29] Jonathan Robie, Ghislain Fourny, Matthias Brantner, Daniela Florescu, Till Westmann, and Markos Zaharioudakis. JSONiq: The JSON Query Language. http://www.jsoniq.org/, 2016.

[30] Luc Segoufin and Cristina Sirangelo. Constant-memory validation of streaming xml documents against dtds. In *International Conference on Database Theory*, pages 299–313. Springer, 2007.

[31] The Apache Software Foundation. Apache CouchDB. http://couchdb.apache.org/, 2015.

[32] The Neo4j Team. The Neo4j Manual v3.0. http://neo4j.com/docs/stable/, 2016.

[33] Karianto Wong and Christof Löding. Unranked tree automata with sibling equalities and disequalities. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, pages 875–887, 2007.