JSON Functionally

Chapter · August 2020

DOI: 10.1007/978-3-030-54832-2_12

1 author:

Jaroslav Pokorný
Charles University in Prague
207 PUBLICATIONS **1,458** CITATIONS

SEE PROFILE

JSON Functionally

Jaroslav Pokorný[0000-0003-3177-4183]

Faculty of Mathematics and Physics, Charles University, Prague,
Czech Republic
pokorny@ksi.mff.cuni.cz

Abstract. Document databases use JSON (JavaScript Object Notation) for data representation. In the category of NoSQL databases they support an ability to handle large volumes of data at the absence of an explicit data schema. On the other hand, schema information is sometimes essential for applications during data retrieval. Consequently, there are approaches to schema construction in the JSON community. We will suppose JSON collections equipped by a schema compatible in its expressivity to the JSON Schema recommendation. We use a formal approach based on typed functional data objects. We accommodate a classical approach to functional databases based on a typed λ-calculus to obtain a powerful non-procedural query language over JSON data with sound semantics.

Keywords: Document databases, JSON, typed lambda calculus, functional data objects, querying JSON data, database integration, MongoDB

1 Introduction

Document databases (DBs), also known as document stores or document-oriented DBs, are one of the main categories of NoSQL DBs. For example, DB-Engines Ranking[1] considers more than 40 document DBs. Document DBs handle collections of objects (called *documents*) represented in hierarchical formats such as XML or JSON. Originally, JSON[2] (JavaScript Object Notation) is a popular data format based on the data types of JavaScript programming language. Similarly to XML it belongs to a category of mark-up languages. Each document is composed of a (nested) set of fields and is associated with a unique identifier for indexing and retrieving purposes. Generally, these systems offer richer query languages than those in other NoSQL categories, being able to exploit the structuredness of the objects they store.

JSON occurs in a number of NoSQL DBs. For example, documents are represented as JSON objects in ArangoDB[3]. OrientDB[4] is a NoSQL solution with a hybrid document-graph engine that adds several compelling features to the document DB model.

[1] http://db-engines.com/en/ranking (retrieved on 22. 5. 2020)
[2] http://www.json.org/ (retrieved on 22. 5. 2020)
[3] https://www.arangodb.com/ (retrieved on 22. 5. 2020)
[4] https://orientdb.com/ (retrieved on 22. 5. 2020)

Also Couchbase[5] uses a JSON-based document store, similarly Apache CouchDB[6] stores JSON documents with the option of attaching non-JSON files to those documents. Here, among document stores, we refer to MongoDB[7], one of the most adopted. Considering such data stores as semistructured DBs, it is relevant to take into account querying such DBs and associated JSON-like query languages. We could cite some special JSON query languages, e.g., JSONiq[8]—a simplified version of XQuery language adapted for JSON.

As it is typical for NoSQL DBs, they are mostly schemaless. On the other hand, a schema-oriented approach is being based on application query patterns. This is departure from RDBMS where schemas are designed for optimizing storage, in case of NoSQL they are designed for access. Hence, NoSQL follows 'Query Driven Design'. Some document DBs offer a possibility to express JSON schema, e.g., in the language JSON Schema [13], [3]. This language is an attempt to provide a general purpose schema language for JSON, so we can optionally enforce rules governing document structures. Authors of [1] identify a JSON type language and design a schema inference algorithm enabling reasonable schema inference for massive JSON data collections.

We will consider just the document DBs with a schema in this paper. The goal is to propose new possibilities for querying JSON data. In our approach, we accommodate some classical approaches to functional DBs, e.g. [4], [8], based on a typed λ-calculus, to obtain a powerful non-procedural query language called JSON-λ with sound semantics. The associated model of JSON data deals with data as with functions. Our goal will be to capture most of the features offered by the functional approach used for modelling XML data and its querying described in [9, 10].

Section 2 presents a short overview of JSON data modelling including the JSON schema format. Section 3 offers a document querying in MongoDB. In Section 4, we describe a functional type system appropriate for typing JSON data and constructing JSON schema. Section 5 introduces a version of a typed λ-calculus for manipulating JSON data and a number of examples of its use. Section 6 concludes the paper.

2 Modelling JSON data

JSON data model describes the basic data structures and semantics of the underlying JSON data [5]. It covers key aspects for JSON data format. JSON defines three major object types for document fields: *object*, *array*, and *value*. An object is an unordered set of name/value pairs, where values can be of type string, number, or other objects. Objects are in {} braces, each name wrapped in "double quotes" is followed by ":", name/values pairs are separated by ",". Arrays are ordered collections of elements (with no constraint on their structure). That is, arrays can mix both basic and complex types. Arrays are bounded by "[" and "]", their elements are separated by ",". A JSON value

[5] https://docs.couchbase.com/home/index.html (retrieved on 22. 5. 2020)
[6] https://couchdb.apache.org/ (retrieved on 22. 5. 2020)
[7] https://docs.mongodb.com/manual/ (retrieved on 22. 5. 2020)
[8] http://www.jsoniq.org/docs/JSONiqExtensionToXQuery/html-single/index.html (retrieved on 22. 5. 2020)

may be either an object, array, number, string, `true`, `false`, or `null`. A *JSON instance* contains a single JSON value.

Clearly, JSON is simpler than XML. For example, it does not consider attributes. Obviously, XML attributes could be also considered as JSON objects, e.g., encoded by prefixing the attribute name with the @ symbol. Similarly to XML, JSON objects could be modelled by trees [2]. To present differences and similarities of XML and JSON data, we use the documents in Fig. 2.1 and Fig. 2.2, respectively.

```
<Book>
  <Title> Fundamentals of Database Systems </Title>
  <Authors>
    <Author> Elmasri,R.</Author>
    <Author> Navathe,S.B.</Author>
  </Authors>
  <Date>2015</Date>
  <Publisher>Pearson</Publisher>
</Book>
```

Fig. 2.1. Data XML-formatted

```
{
  "Book":
    {
      "Title": "Fundamentals of Database Systems",
      "Authors":["Elmasri,R.", "Navathe,S.B."],
      "Date":"2015",
      "Publisher":"Pearson"
    }
}
```

Fig. 2.2. The same data as in Fig. 2.1 JSON-formatted

The ability to store nested or hierarchical data within a text file structure makes JSON a powerful format to use as we are working with larger text datasets.

2.1 JSON Schema

JSON schema is a format that may be used to formalize constraints and requirements to JSON files. The language JSON Schema[9], similarly to XML Schema, SQL-like relational schemas, etc., is a powerful tool for validating the structure of DB data. However, it is different from a similar notion used in XML data model. i.e., XSD. A *JSON schema* is a JSON document, and that document must be an object. Object members (or *properties*) defined by JSON are called *keywords*, or *schema keywords*. With JSON Schema we can specify what properties exist, which type(s) can be used, if properties

[9] https://json-schema.org/ (retrieved on 22. 5. 2020)

are required and how the JSON document is composed. The JSON schema in Fig. 2.3 describes a more complex `name` type and an associated object.

```
{                                    {
 "name":{                            "name":{
  "type":"object",                     "firstname":"Ramez",
  "properties":{                       "surname":"Elmasri"
   "firstname":{"type":"string"},            }
   "surname":{"type":"string"}     }
}}}
```

Fig. 2.3. JSON schema fragment and related JSON data

It is also best practice to include an $id property as a unique identifier for each schema. We can set it to a URL at a domain, e.g.,

```
{"$id": "http://mydomain.com/schemas/myschema.json"}
```

Also some basic integrity constraints (ICs) are allowed to formulate as it is shown in Fig. 2.4 (without `firstname` and `surname`).

```
{
  "$schema":"http://json-schema.org/draft-04/schema",
  "type":"object",
  "properties":{
     "book":{
        "type":"object",
        "properties":{
           "title":{"type":"string"},
           "authors":{"type":"array",
                      "minItems":1,
                      "maxItems":3,
                      "items":{"type":"string"}},
           "issued":{"type":"number"},
           "required":["title","authors"],
           "additionalProperties":false
                      }
            }  },
      "required":["book"],
      "additionalProperties":false
}
```

Fig. 2.4. JSON schema fragment with ICs

Semantics of objects and elements of arrays of a JSON schema instance is straightforward. For example, an object instance is valid against `minItems` and `maxItems` if its number of authors is from interval <1,3>, etc. The `type` of authors' items in this array is `string`. The first restriction `additionalProperties` specifies if the array cannot contain items which are not specified in the schema.

3 Querying JSON data in MongoDB

MongoDB instance is composed of a set of named *collections* of BSON documents, nested up to an unbounded depth. BSON (Binary Serialized dOcument Notation) is a binary serialization of JSON, with which such documents share the schemaless, arbitrarily nested structure. A collection may have zero or more documents. Documents are addressed in the DB via a unique key `_id`. Documents within a collection can have different fields. MongoDB also supports data types that are not part of original JSON specification, such as `Date, Timestamp`, `Binary`, `ObjectID`, `RegExp`, `Long`, `Undefined`, `DBRef`, `Code`, and `MinKey`. This is known as Extended JSON.

From Version 3.6, MongoDB supports JSON Schema and provides a JSON Schema Validator from within the application that runs like a trigger to check any changes to the data via inserts or alterations. In the next example, a document key is not included. Notation is simplified a little, e.g. `""` braces are omitted for object names.

```
{name:{firstname:"John", lastname:"White"},
address:"Mostecka 25, 118 00 Prague 1",
grandchildren:["Klara","Magda","Richard"],
age:23
}
```

Default DB of MongoDB is 'db'. A query is specified using the `find` command, e.g.

```
db.<collection>.find({<age>:{$in:[<value>,
                                  <value>]}})
```

specifies that the `age` of persons documents should be one of 2 values, e.g.,

```
db.persons.find({age:{$in:[23,25]}})
```

Comparison operators like `$lt, $lte, $gt, $gte, $ne` are allowed. They correspond to <, <=, >, >=, and ≠, respectively. More conditions separated by "," represent a conjunction. For example,

```
db.persons.find({age: 23, grandchildren:["Magda"]})
```

Another option is to use `$exists`.

```
db.<collection>.find({<name>:{$exists: true}})
```

The query searches for documents with an existing field.

We can observe that the above examples enable to express some forms of AND queries. Explicit `$or` enables to express OR queries. More advanced retrieval uses querying on embedded documents (in MongoDB terminology) using paths in JSON graph.

In Sections 4-5 we will propose a calculus-oriented approach to querying JSON data, which requires a JSON schema expressed by a set of typed functions. This language JSON- λ will be more powerful than querying allowed by MongoDB.

4 Typing JSON data

In our approach, JSON uses typed objects. To include typed objects into a DB processing, we introduce a set **O** of *abstract objects*. The set **O** serves as a reservoir for

construction of JSON objects. An empty abstract object ε is also the member of **O**. The *content* of an abstract object will be either a string from `STRING`, in the easiest example, or a group of (sub)objects, or empty. There is difference between the empty object ε and empty content of an empty object.

Consider now, e.g., the object `{"Date":"2015"}`. It is a JSON instance of the `Date` object, which will be conceived as a (partial) function from **O** into `STRING`. For an `o`∈**O**, `Date(o)` returns the date 2015. The `Date` function is changing in time similarly as relations during the life of a relation DB. Similarly, the `name` type from Example 2.1 can be conceived as a set of two functions from **O** into **O** (including abstract first names and surnames). The current `name` object, i.e. the one stored in a given JSON DB, is a function assigning to each `o`∈**O** at most a couple of abstract objects from **O**.

4.1 Typing JSON objects

Summarizing the notions given above, we will distinguish among object types and objects. *Objects* are of object types. Because object types are functions, objects are their values. Obviously, the values need mark-up given by the name of the object. Actually, a presentation of objects is done by real JSON objects.

First, we introduce a general functional type system supporting a structure of JSON data.

Definition 4.1. The existence of some (*primitive*) *types* $S_1,...,S_k$ ($k \geq 1$) is assumed. They constitute a *base* **B**. More complex types are obtained in the following way:

(a) Every member of the base **B** is a (*primitive) type* over **B**.
(b) If T_1, T_2 are types over **B**, then $(T_1 \rightarrow T_2)$ is a (*functional) type* over **B**.
(c) If $T_1,...,T_n$ ($n \geq 1$) are types over **B**, then $\{T_1,...,T_n\}$ is a *set type* over **B**.
(d) If $T_1,...,T_n$ ($n \geq 1$) are types over **B**, then $[T_1,...,T_n]$ is an *array type* over **B**.

The *type system* **T** *over* **B** (or **T** if **B** is assumed) is the least set containing types given by (a)-(d).

If members of **B** are interpreted as mutually disjoint non-empty sets, then $(T_1 \rightarrow T_2)$ denotes the set of all (total or partial) functions from T_1 into T_2. An object *o* of the type T is called a *T-object* and can be denoted *o*/T. $\{T_1,...,T_n\}$-objects contain an unordered set of respective $T_1,...,T_n$ objects. $[T_1,...,T_n]$-objects are arrays of respective $T_1,...,T_n$ objects.

For example, `NUMBER`, `STRING`, `BOOL` ∈ B. The type `BOOL` is defined as the set `{TRUE, FALSE}`. It allows typing some objects as sets and relations. They will not considered here as separate units. Both can be modelled as unary and n-ary characteristic functions, respectively. Thus, both notions are redundant in **T**.

For example, mathematical functions may be easily typed. Arithmetic operations +, -, *, / are examples of ((`NUMBER`, `NUMBER`) → `NUMBER`)-objects. Logical connectives, quantifiers and predicates are also typed functions: e.g., **and**/((`BOOL`, `BOOL`) → `BOOL`), R-identity $=_R$ is ((R,R) → `BOOL`)-object, universal R-quantifier Π_R, and existential R-quantifiers Σ_R are ((R→ `BOOL`) → `BOOL`) - objects. As for our notational convention, we use an infix notation for logical functions. Similarly, we write '∀*x*...' and '∃*x*...' for

application of the universal and existential quantifier, respectively. With **T**, it is possible to type functions of functions, nested tables, ISA-hierarchies, etc.

4.2 Typing JSON regular expressions

First, in the type system $\mathbf{T}_{reg}$ we will describe regular expressions over character data and named character data. We start with the base **B** containing primitive types `BOOL`, `NUMBER` and `STRING`. Moreover, outside **B** we use a set `NAME` of strings. Obviously, one type of strings would be enough, but we want to distinguish between names of objects and the most important part of objects content - character data.

Our regular expressions will be restricted comparing to regular expressions in XML [9]. The reason is that in a set type keys of $T_1,...,T_n$ must be unique. When the names within a set object are not unique, the behaviour of software that receives such an object would be unpredictable.

Definition 4.2. Let **B** = {`STRING`, `NUMBER`, `BOOL`}, a set `NAME` of names and **T** the type system over **B**. Then we define the *type system* $\mathbf{T}_{reg}$ over **B** as follows.

1. Every member of the base **B** is an (*primitive) type* over **B**.
2. Let name ∈ `NAME`. Then
   name: `STRING` is an (*elementary*) *type* over **B**, */named character data/*
   name: is an (*empty elementary) type* over **B**.
3. Let T be a set type or named character data. Then
   T* is a *type* over **B**. */zero or more/*
   T+ is a *type* over **B**. */one or more/*
   T? is a *type* over **B**. */zero or one/*
4. Let T be a type given by a step 3. Let name ∈ `NAME`. Then name: T is a *type* over **B**. */named type/*

Restrictions:

- Let $[T_1,...,T_n]$ be an array type. Then any T_i can be T*, T+, or T?.
- Let $\{T_1,...,T_n\}$ be a set type. Then any T_i can be T?.

The *type system* $\mathbf{T}_{reg}$ *over* **B** (or $\mathbf{T}_{reg}$ if **B** is understood) is the least set containing types given by (1)-(4).

Then name:`STRING` denotes the set of character data named name, and name: denotes the empty character object. T? denotes the set of objects of type T∪ `NIL`.

Now we will define JSON object types. We suppose for each name used in name:`STRING` or in name: the existence of the NAME name denoting an associate object type. The same holds for each name ϵ `NAME`.

Definition 4.3. Let $\mathbf{T}_{reg}$ over **B** be the type system from definition 4.2 and **O** be the set of abstract objects. Then the *object type system* $\mathbf{T_O}$ *induced by* $\mathbf{T}_{reg}$ (or $\mathbf{T_O}$ if $\mathbf{T}_{reg}$ is understood) is the least set containing the object types given by the following rule:

- Let name:T be from $\mathbf{T}_{reg}$. Replace all names in name:T by their upper-case version. Then NAME:T is a member of $\mathbf{T_O}$.

The functional semantics of object types from $\mathbf{T_O}$ associates with NAME:`STRING` the set of all functions from **O** into name:`STRING`. For a non-elementary type T from $\mathbf{T_O}$, the semantics of NAME:T is also functional, but the functions are more complex.

Let B, C, and D be elementary object types, i.e. they denote the function space $\mathbf{O} \rightarrow$ STRING. Then e.g. A:{C, D?} denotes a function space

$\mathbf{O} \rightarrow (\mathbf{O} \rightarrow \mathtt{STRING}) \times (\mathbf{O} \rightarrow \mathtt{STRING} \rightarrow \mathtt{BOOL})$.

Note that $(\mathbf{O} \rightarrow \mathtt{STRING} \rightarrow \mathtt{BOOL})$ abbreviates $((\mathbf{O} \rightarrow \mathtt{STRING}) \rightarrow \mathtt{BOOL})$. Due to the properties of Cartesian product we can easily work with its components. For example, if A:{B, C, D} is an object type with A, B, C, and D as above, then A.B denotes functions of type $(\mathbf{O} \rightarrow \mathbf{O} \rightarrow \mathtt{STRING})$. Note, that it is the same space as for A:{B}. Let there is an object of this type. Then for each A abstract object we obtain an B abstract object and then B's character data. We observe that with names it is not necessary to use any positional notation.

4.3 Toward a JSON functional schema

We use a part of the JSON schema in Fig. 2.4 and one valid book object in Fig. 4.1, i.e., a simple BIBLIO DB is at disposal. The FIRSTNAME:STRING denotes the set of all functions associating with each abstract object from **O** at most one object of type firstname:STRING. The object of this type stored in the BIBLIO DB in Fig. 4.1 is defined for two abstract objects with values 'Anthony' and 'Joe', respectively. Obviously, in other BIBLIO DB the object of type FIRSTNAME could be defined also for abstract objects associated with no book object. There are people (authors) who are not book authors. Object types from $\mathbf{T_O}$ related to the book object in Fig. 4.1 include

TITLE:STRING
FIRSTNAME:STRING
SURNAME:STRING
LOCALITY:STRING
ZIP:STRING
ISSUED:NUMBER
ADDRESS:{LOCALITY, ZIP}
BOOK:{TITLE, AUTHORS, ISSUED?} (1)
AUTHORS: [AUTHOR, AUTHOR]
NAME:{FIRSTNAME, SURNAME}
AUTHOR: {NAME, ADDRESS?}

In $\mathbf{T_O}$ we could specify, e.g., the BOOK object type by one complex expression as

```
BOOK:{TITLE:STRING,
 AUTHORS:[AUTHOR:{NAME:{FIRSTNAME:STRING,  SURNAME:STRING},
                  ADDRESS?{LOCALITY:STRING, ZIP:STRING}},
          AUTHOR: {NAME:{FIRSTNAME:STRING, SURNAME:STRING},
                   ADDRESS?{LOCALITY:STRING,  ZIP:STRING}
                  }],
 ISSUED?:NUMBER}                                              (2)
```

JSON DB contains only one BOOK object in this example. In the case of NAME object type, the associated function is defined on two abstract objects. Obviously, NAME in this case means something else than NAME in Definition 4.3.

```
{
  "book":{
    "title":"Business objects",
    "authors":["author":{
                  "name":{
                    "firstname":"Anthony",
                    "surname":"Newman"
                          }
                  "address":{
                    "locality":"Malostranske 25, Praha",
                    "ZIP":"118 00"
                            }
                          },
                "author":{
                  "name":{
                    "firstname":"Joe",
                    "surname":"Batman"
                          }}]
          }}
```

Fig. 4.1. JSON document containing a `book` object

To consider books with exactly two authors is rather restrictive. We could take into account the variant AUTHORS:[AUTHOR+], i.e. books with one or more authors. Then the expression (2) would be much simpler.

Obviously, abstract objects are not visible. Intuitively, we can imagine objects stored in a JSON DB, which is described by appropriate typed object variables. A valuation of these variables provides a *DB state*. In practice, we could consider JSON DBs containing one JSON object of type BOOKS[BOOK+] containing books of type BOOK.

Thus, we will conceive a *JSON-database schema,* $\mathbf{S}_{JSON}$, as a set of variables of types from $\mathbf{T}_O$. Given a database schema $\mathbf{S}_{JSON}$, a *JSON-database* is any valuation of variables in $\mathbf{S}_{JSON}$. For convenience, we denote the variables from $\mathbf{S}_{JSON}$ by the same names as names from $\mathbf{T}_O$, e.g. BOOK, AUTHOR, etc. Then object types in (1) represent a JSON-database schema. Explicit ICs on the JSON-database are not considered in this paper.

5 Querying JSON data with λ-terms

Recent query languages over JSON data have roots in languages for querying semistrutured data. There the most important query languages deal with XML data. They enable easily to apply the notion of path in the query expressions as well as regular path expressions. We use this strategy here, too.

Due to the functional features of $\mathbf{T}_O$, our approach will be based on the typed λ-calculus extended with arrays. The λ-calculus serves as a manipulation tool which directly supports manipulating objects typed by $\mathbf{T}_O$.

First, we introduce a general definition of λ-terms (shortly terms).

Definition 5.1 Starting with a collection **Func** of constants, each of a fixed type, and denumerably many variables for each type from **T**, the *language of terms* (LT) is inductively defined as follows. Let types T, T_1, ..., T_n ($n \geq 1$) are members of **T**.

1. Every variable of type T is a *term* of type T.
2. Every constant (a member of **Func**) of type T is a *term* of type T.
3. If M is a term of type $((T_1,...,T_n) \rightarrow T)$ and $N_1,...,N_n$ are terms of type $T_1,...,T_n$, respectively, then
   $M(N_1,...,N_n)$ is a *term* of type T. */application/*
4. If $x_1,...,x_n$ are distinct variables of types $T_1,...,T_n$, respectively, and M is a term of type T, then
   $\lambda x_1,...,x_a(M)$ is a *term* of type $((T_1,...,T_n) \rightarrow T)$. */λ-abstraction/*
5. If M is a term of type $\{T_1,...,T_n\}$ and N is of type $T \in \{T_1,...,T_n\}$. Then
   $N(M)$ is a *term* of type T. */set element/*
6. If $N_1,...,N_n$ are terms of types $T_1,...,T_n$, respectively, then
   $[N_1,...,N_n]$ is a *term* of type $[T_1,...,T_n]$. */array/*
7. If M is a term of type $[T_1,...,T_n]$, then
   $M[1],...,M[n]$ are *terms* of types $T_1,...,T_n$, respectively. */elements of array/*

Terms can be interpreted in a standard way by means of an interpretation assigning to each constant symbol from **Func** an object of the same type, and by a semantic mapping from LT into all functions and Cartesian products given by the type system **T**. In short, we assume a standard 'fixed' interpretation in which an application is evaluated as the application of the associated function to given arguments, a λ-abstraction 'constructs' a new function of the respective type, etc.

5.1 λ-calculus and JSON data

We restrict our λ-calculus to JSON data, i.e. we will suppose the type systems $\mathbf{T}_{reg}$ and $\mathbf{T_O}$ induced by $\mathbf{T}_{reg}$. To add some expressiveness to LT, we permit some useful Boolean functions such as comparing predicates =, <, ≤, …, quantifiers, and logical connectives in **Func**. We will extend slightly the notion of variable. To achieve querying tags, it is necessary to consider not only variables typed by $\mathbf{T_O}$. We need also variables like x:y, where x/NAME and y/T with T from $\mathbf{T}_{reg}$ or $\mathbf{T_O}$. Obviously, the resulted type system and the language remain still very general, i.e. they are not *restricted* in the sense of permitting only JSON-data as the query output. For example, we will show that also relational data can be extracted from JSON data.

It is easily verified that terms of LT are usable to expressing queries. Each term defines a function after its evaluation over a given DB. For queries we use applications of functions and λ-abstractions. A typical *query term* is of form

λ .. (λ .. …(expression)…),

where expression is of type BOOL. A query term consists of two parts: the query part (here expression) and a constructor part (here the lambda part of the query term). A significant role is assigned to the lambda part. Similarly to XML query languages, we can construct the answer to a query, e.g., a new JSON document with a rich nested structure.

Another possibility of querying, as we shall see, is based on application terms. We examine now some terms and queries. We will use variables, e.g., *e*/**O**, other letters will denote abstract objects. Functional semantics of objects allows specifying paths via compositions of functions. For example, for a book abstract object *b* we can get the first author's surname for the book associated with *b*.

SURNAME(NAME(AUTHORS(b)[1]))

The variables AUTHORS, NAME and SURNAME from $\mathbf{S}_{JSON}$ are valuated by actual objects of respective types. Then this application returns character data, i.e. the value of a SURNAME object. To simplify the notation, we can use a more convenient "path-like" notation

b.AUTHORS[1].NAME.SURNAME

Then we can easily use comparisons like

b.AUTHORS[1].NAME.SURNAME = 'Newman'

5.2 Querying JSON data—examples

The first feature of most JSON-like query languages is the matching data using patterns. Consider again book's documents with one entry in Fig. 4.1 and the query D1: Find the addresses of the book author Anthony Newman. In JSON-λ language the query is expressed as follows:

D1(JSON-λ):

λx (.BOOK.AUTHOR.NAME.SURNAME = 'Newman' **and**
.BOOK.AUTHOR.FIRSTNAME = 'Anthony' **and**
.BOOK.AUTHOR.ADDRESS = x)

Similarly to the JSON style, it is possible to omit parts of paths. We are using "." for specification root objects. The dot starting all three paths expresses that the path begins in an abstract book object. The object is common for all three paths. There is an equivalent expression that uses an existential quantifier and *e* variable for this purpose.

Obviously, *x*/**O**. We can derive this fact from the context of *x* in the body of λ-abstraction. Each evaluation is running over all values of *x*, because *x* is free variable in the λ-abstraction body. Thus, as a result, we obtain a set of **O**-objects in this case, which is not very user friendly. The set will be a singleton in the case when there is only one author Anthony Newman in the BIBLIO DB. Of course, for real addresses, the LOCALITY and ZIP would be necessary. Two authors Anthony Newman (even with the same address) could be assigned to two different abstract objects. To obtain the content of an abstract object we introduce notation of underlining. Then $\underline{x}$ returns the content of the abstract object valuating *x*. Replacing *x* by $\underline{x}$ in query term of D1(JSON-λ), we could obtain a presentation of the result like

```
{ "locality":"Malostranske 25, Praha",
  "ZIP":"118 00"
}
```

i.e. the result is a JSON object it this case. To obtain better semantics we should write

λ address:x(…

in the lambda part of D1(JSON-λ). In this case, the choice of JSON names in the result can be user-controlled explicitly. This reminds introducing new XML tags (opening and closing) into a result of an XQuery query given by its RETURN clause.

We can also require fair `STRING` objects. For example, the query D2: Find all addresses, we formulate in JSON-λ as
D2(JSON-λ):

λx, y (.ADDRESS.LOCALITY = x **and** .ADDRESS.ZIP = y)

After the query evaluation, we obtain a set of couples like (`"Malostranske 25, Praha", "11800"`), etc. That is, the result is a binary relation it this case.

In more convenient notation, we can group logical conditions belonging to the same path in D1(JSON-λ). Then, we can write

λx (.BOOK.AUTHOR.NAME.(SURNAME = 'Newman'
and FIRSTNAME = 'Anthony') **and** .BOOK..ADDRESS = x)

The descendant operator ".." indicates any number of intervening levels. Of course, it's just an abbreviation for a more complex expression. Then the lambda part of D1(JSON-λ) can be even shortened to

λx (.BOOK..NAME.(SURNAME = 'Newman'
and FIRSTNAME = 'Anthony') **and** .BOOK..ADDRESS = x)

According to the DB notions, till now discussed JSON-λ features represent a *selection* of data based on simple logical conditions. The term

λx (.BOOK..NAME.(SURNAME = 'Newman' **and** FIRSTNAME = 'Anthony')
and .BOOK..ADDRESS = x **and** .BOOK.ISSUED > 2015)

ensures to select the addresses of authors whose books were published after 2015.

The following powerful feature of JSON-λ simulates the *object join operation* over a common value. Moreover, we will show how this operation combines data from different JSON DBs. The second JSON DB of our example, ADDRESSBOOK, contains records from an address book. Its entry in Fig. 5.1 contains only data about one person.

```
{
  "addressbook":{
          "person":{
              "ID":"111-22-3333",
              "surname":"Newman",
              "name":"Anthony"
              "titled":"Dr. A. Newman",
              "address":"Malostranské 25, Praha 1",
              "links":["tel":"2191 4268",
                       "tel":"2191 4323",
                       "email":"newman@ksi.mff.cuni.cz"]
                 } }
}
```

Fig. 5.1. JSON document containing an `addressbook` object

We try to join objects from both DBs in the query D3.

D3: Who from our contacts are authors of books? Give also their e-mail addresses.

We need to distinguish between two data sources, e.g. using a prefix notation.

D3(JSON-λ):

λ x, y, m (BIBLIO:.BOOK..NAME.(SURNAME = x **and** FIRSTNAME = y) **and** ADDRESSBOOK:.PERSON.(SURNAME = x **and** NAME = y **and** LINKS[3] = m))

The result is a ternary relation of `STRING` data. To obtain the answer in more JSON-like style, we can replace the lambda part of the query term, e.g., by

λ `e-contact`:[`surname`:x, `name`:y, `email`:m]

i.e. the result will be a set of E-CONTACT objects composed from triples of other objects. Because this is a part of a query expression, it is not necessary to solve the problem of assigning new abstract objects to parts of the result.

In general, a constructing JSON data in the answer is ensured by the lambda part of a query term (see, D3(JSON-λ)). A „flattening" of a hierarchy is also possible to do. It reminds a denormalization of relations. For example, we can easily formulate a query in JSON-λ, which produces a list of triples (`title`, `firstname`, `surname`). An author can occur in the result many times, dependent on the number of his/her books.

We can write more structured queries in JSON-λ.

D4. For each author find the book titles, where the author is the first author.

D4(JSON-λ):

λx.NAME.FIRSTNAME, `surname`:x.NAME.SURNAME
(λ`title`:y(BOOK.(AUTHORS[1](x) **and** TITLE = y)))

D5. Find the names of authors for each book.

D5(JSON-λ):

λ`title`:y (λ`name`:x.NAME (∃i .BOOK..(AUTHORS[i](x) **and** TITLE = y)))

We have used here the NAME component of the variable *x*. Names will contain properties `firstname` and `surname`.

Another category of queries could use the universal quantifier and implication similarly to, e.g., the domain relational calculus. We conclude the description of JSON-λ with usage of aggregate functions in queries. Suppose now the query

D6. For each book, find the number of its authors.

D6(JSON-λ):

λ x, n (.BOOK..(TITLE = x **and** COUNT(AUTHORS) = n))

The query returns a relation whose each row is composed from a book title and associated number of its authors. Suppose here AUTHORS:[AUTHOR+] in JSON-database schema (1). The type of *n* variable is derived from COUNT type. Remind that COUNT assigns to each set or array its cardinality.

6 Conclusions

In this paper, we have shown a DB-oriented view on JSON data based on a functional approach to typing. JSON schema is a set of typed functions. Then, a version of typed

λ-calculus (LT language) was used as a formal background for querying JSON data. Comparing to MongoDB querying, LT is more expressive due to logical connectives, quantifiers, arithmetic operations, and aggregation functions included in **Func**. Query answers can be relations, tree structures of `STRING` values, and even new JSON data. Queries in MongoDB return only subsets of a collection. Because the proposed LT language is very general, it would be necessary to use only its appropriate subset in practice.

With the LT language we can do even an integration of NoSQL and relational DBs [11-12]. Integration of relational and JSON data can be done without problems.

There is also a standardized approach to integration of SQL and JSON in the SQL:2016 [6]. There the SQL/JSON path language and the SQL/JSON operators are defined. More about JSON integration with relational DBs can be found in [7].

Acknowledgments. This work was supported by the Charles University project Q48.

References

1. Baazizi, M.A., Lahmar, H.B., Colazzo, D., Ghelli, G., Sartiani, C.: Schema Inference for Massive JSON Datasets. In: Proc. of EDBT 2017, pp. 222-233 (2017).
2. Bourhis, P., Reutter, J.L., Vrgoč, D, Suárez, F.: JSON: Data model and query languages. Information Systems 89: 123-135 (2020).
3. Droettboom, M., et al: Understanding JSON Schema Release 7.0. Space Telescope Science Institute (2019).
4. Duzi, M., Pokorny, J.: Semantics of General Data Structures. In: Information modelling and knowledge bases IX. P. –J. Charrel, H. Jaakkola, H. Kangassalo (Eds.), pp. 115-130, IOS Press, Amsterdam, Netherlands (1998).
5. Lv, T., Yan, P., He, W.: Survey on JSON Data Modelling. Journal of Physics: Conference Series, Vol. 1069 (2018).
6. Michels, J., Keith, H., Kulkarni, K., Zuzarte, C., Liu. Z., H., Hammerschmidt, B., Zemke, F.: The New and Improved SQL:2016 Standard. SIGMOD Record, Vol. 47, No. 2, pp. 52-60 (2018).
7. Petković, D.: JSON Integration in Relational Database Systems. International Journal of Computer Applications (0975 –8887), Vol. 168 –No.5, pp. 14-19 (2017).
8. Pokorny, J.: Database semantics in heterogeneous environment. In: Proc. of 23rd Seminar SOFSEM'96: Theory and Practice of Informatics, K.G. Jeffery, J. Král, M. Bartošek (Eds.), pp. 125-142, Springer-Verlag (1996).
9. Pokorný, J.: XML functionally. In: Proc. of IDEAS2000, B. C. Desai, Y. Kioki, M. Toyama (Eds.), IEEE Comp. Society, pp. 266-274 (2000).
10. Pokorný, J.: XML-λ: an Extendible Framework for Manipulating XML Data. In: W. Abramowicz (Ed.) BIS 2002, Poznan, pp. 160-168 (2002).
11. Pokorný J.: Integration of Relational and NoSQL Database. In: N. T. Nguyen, et al (eds), ACIIDS (2), LNCS, vol. 10752, pp. 35-45, Springer (2018).
12. Pokorný J.: Integration of relational and graph databases functionally. Foundations of Computing and Decision Sciences 44 (4), 427-441 (2019).
13. Pezoa, F., Reutter, J.R., Suarez, F., Ugarte, M., Vrgoč, D.: Foundations of JSON Schema. In: Proc. of WWW 2016, pp. 263-273 (2016).