Noname manuscript No.
(will be inserted by the editor)

Parametric Schema Inference for Massive JSON Datasets

Mohamed-Amine Baazizi · Dario Colazzo · Giorgio Ghelli · Carlo Sartiani

the date of receipt and acceptance should be inserted later

Abstract In recent years, JSON established itself as a very popular data format for representing massive data collections. JSON data collections are usually schemaless. While this ensures several advantages, the absence of schema information has important negative consequences as well: data analysts and programmers cannot exploit a schema for a reliable description of the structure of the dataset, the correctness of complex queries and programs cannot be statically checked, and many schema-based optimizations are not possible.

In this paper we deal with the problem of inferring a schema from massive JSON datasets. We first identify a JSON type language which is simple and, at the same time, expressive enough to capture irregularities and to give complete structural information about input data. We then present our contributions, which are the design of a parametric and parallelizable schema inference algorithm, its theoretical study, and its implementation based on Spark, enabling reasonable schema inference time for massive collections. Our algorithm is parametric as the analyst can specify a parameter determining the level of precision and conciseness of the inferred schema. Finally, we report about an experimental analysis showing the effectiveness of our approach in terms of execution time, conciseness of inferred schemas, and scalability.

Keywords JSON, schema inference, map-reduce, Spark, big data collections

Mohamed-Amine Baazizi
Sorbonne Université, LIP6
E-mail: mohamed-amine.baazizi@lip6.fr

Dario Colazzo
LAMSADE - Université Paris Dauphine, PSL Research University, CNRS, LAMSADE, 75016 Paris, France
E-mail: dario.colazzo@dauphine.fr

Giorgio Ghelli
Dipartimento di Informatica, Università di Pisa, Italy
E-mail: ghelli@di.unipi.it

Carlo Sartiani
DIMIE - Università della Basilicata
E-mail: sartiani@gmail.com

1 Introduction

Big Data applications typically process and analyse very large structured and semi-structured datasets. In many of these applications, especially those relying on NoSQL document stores, data are represented in JSON (Java Script Object Notation) [15], a data format that is widely used thanks to its flexibility and simplicity.

JSON data collections are usually schemaless. This ensures several advantages: in particular it enables applications to be quickly deployed without waiting for a schema to be specified, and makes them resilient to data irregularity. Unfortunately, the lack of a schema makes it impossible to statically detect any mismatch between the actual structure of data and the structure expected by complex queries and programs; furthermore, programmers cannot use a schema description to ease the production of correct code, and schema-based optimizations are not possible.

JSON datasets may be retrieved from remote, uncontrolled sources, with no schema information, but they may also be generated by applications whose code is known. In these cases some knowledge is available about the structure of the program output, but, when the code starts being complex, schema inference is still quite useful. In some other cases, remote JSON sources can be accessed by APIs (e.g., Twitter APIs) that sometimes are provided with some schema descriptions. Un-

fortunately, these descriptions are often imprecise and incomplete.

In this paper we deal with the problem of inferring a schema from massive JSON datasets. Our main goal in this work is to infer structural properties of JSON data, that is, a description of the structure of JSON objects and arrays that takes into account nested values, optional keys, and any other kind of structural variations. These are the main properties that characterize semi-structured data, and having a tool that ensures *fast*, *precise*, and *concise* inference is crucial in modern applications characterized by agile consumption of huge amounts of data coming from multiple and disparate sources.

It is worth stressing that, even if in some cases a JSON dataset has a rather regular structure, the only way for a user to be sure that all possible (optional) fields are identified is to explore the entire dataset either manually or by means of scripts that must be manually adapted to each particular JSON source, with weak guarantees of efficiency and soundness. Our approach, instead, applies to any JSON data collection, and is shown to be sound and to be effective on massive datasets.

The approach we propose here is based on a JSON schema language and on an efficient, parametric, inference algorithm. The schema language is able to capture detailed structural information about input data despite the presence of any irregularity, and can express that information at different levels of abstraction. This language resembles and borrows mechanisms from existing proposals [31], but it has the advantage to be simple yet very expressive.

The algorithm is designed for an optimized map-reduce implementation, in order to be applicable to massive datasets. It is based on the parallel extraction of a schema for each data item and on the fusion of those schemas that are equivalent during the reduce phase, according to an equivalence relation, which is a parameter of the algorithm. The equivalence relation specifies which properties types must enjoy to be fused together, e.g., one can decide to fuse record types having exactly the same set of labels, or to fuse record types sharing just a common core of fields; hence, a different equivalence relation leads to a difference balance of precision and compactness. We will prove that, whichever equivalence is chosen, the resulting fusion function enjoys the commutative and associative properties, enabling local aggregation in a map-reduce setting, which is crucial for an efficient execution.

The parametrization is a central feature of our approach. In this paper we present some different equivalence relations, to illustrate the flexibility of the approach, and focus on the two equivalences that have the maximal practical interest. These equivalences differ in the way record types are fused. While the first one fuses any two record types, by marking as 'optional' those fields that are not mandatory in both, the second one only fuses record types that share the *same* set of labels. So, while in the first case we obtain very compact schemas, in the second case we obtain a schema that is potentially much bigger, but where field correlation information is preserved.

In a typical scenario, a programmer or data analyst will first run the most compact version in order to gain a general view of the data structure, and will later use a less abstract version in order to get a more complete knowledge of the structural variations. When the first equivalence is used, the resulting schema has usually a size that is small enough to enable a user to consult it in a reasonable amount of time, in order to get a global knowledge of the structural and type properties of the JSON collection, while the second equivalence may generate, depending on the regularity of the data, a schema that is quite bigger.

The generated schema is in any case *path-covering*, in the sense that each path that can be traversed in the tree-structure of the input JSON value can be traversed in the inferred schema as well. This property is crucial to enable a series of query optimization tasks. For instance, thanks to this property JSON queries [3, 12] can be optimized at compile-time by means of schema-based path rewriting and wildcard expansion [27] or schema-based projection [8, 10]. These optimizations are not possible if the schema hides some of the structural properties of the data, as happens in related approaches [35].

Even in its most compact version, our inferred schema precisely captures the presence of optional and mandatory fields in a collection of JSON records, so that the user has already a clear knowledge about i) all possible fields of records, ii) a distinction between optional and mandatory ones. When the schema is expanded to a more precise version, the user can also know which sets of optional fields do, or do not, co-occur in some records.

Our Contribution Our contribution is the design and experimentation of a schema inference algorithm for JSON values that is: parametric; based on a formal specification; designed for efficient map-reduce implementation.

Our schema inference approach consists of two main steps. In the first one, an input collection of JSON values is processed, in parallel, by a Map transformation in order to infer a simple type for each value. The resulting

output is processed by a *type reduction* phase, implemented as a Reduce action, which *fuses* inferred types that have a similar structure. This phase is guided by an equivalence relation (an *ER*) that determines when two types are similar enough to be fused, and when they are too distant and should hence be kept separated. Our type system is flexible enough to allow the same data collection to be described at different levels of abstraction, either by a precise description that may be quite big or by more abstract descriptions, that are smaller but less precise, and the choice of the ER influences this size-precision tradeoff.

In this work we present some ERs and analyze two of them: the first one fuses any two types of the same kind (the kind of a type being *record*, *array*, *integer*, etc.), while the second one is finer in the sense that only record types sharing the same labels are fused.

We prove that the parametrized algorithm is always sound, meaning that, for any ER, the inferred type is always a correct type of the input collection.

We also prove that the Reduce phase enjoys commutativity and associativity. Associativity is also important to enable incremental evolution of the inferred schema under updates: when new values are added to the collection, the type inferred for them can just be combined with the type of the previous values, in order to get the new result.

Our last contribution consists of an Apache Spark [2] implementation of the proposed approach with an experimental evaluation validating our claims of succinctness and efficiency, and a discussion about precision.

1.7 *Remark 1* A preliminary version of this paper appeared in [7]. Compared to that conference version, here we present the following four major additions:

- In [7] we only considered a fusion function driven by
1.25 the kind equivalence. Here, we present a parametric system that can exploit different ERs, and focus our attention on two equivalences of practical interest.
- We provide in the Appendix the proofs for the theorems of soundness, commutativity, and associativity of fusion.
- We describe here several extensions that are of both theoretical and practical interest, and outline how our formal system should be modified in order to fit these extensions (Section 8).
- In [7] we provided a very sketchy experimental evaluation. Here, we provide a wider evaluation involving more schema fusion algorithms and bigger data sets.

Paper Outline The paper is organized as follows. In Section 2 we give an overview of our approach. In Section 3 we survey existing related works. In Section 4, we describe the data model and the schema language we use here, while in Sections 5 and 6 we present our type reduction and schema inference approaches, whose correctness, commutativity, and associativity are proved in Appendix A. In Section 7 we present the results of our experimental evaluation. In Sections 8 and 9, finally, we discuss some possible extensions and draw our conclusions.

2 Overview

In this section we illustrate our approach through an example. To this end, we first briefly recall the general syntax and semantics of JSON values. JSON values are either *atomic* (or *basic*) values, which can be numbers (e.g., 123), strings (e.g., "abc"), booleans (i.e., true/-false), and null, or *complex* values, which can be either unordered sets of key/value pairs called *records* or ordered lists of values called *arrays*. Complex values can be arbitrarily nested and arrays can mix values of different types. The only constraint that JSON values must obey is key uniqueness within each record.

A sample JSON record is illustrated in Figure 1. Syntactically, records use the conventional curly braces symbols whereas arrays use square brackets; finally, string values and keys are wrapped inside quotes in JSON, although we will avoid quotes around keys in our types and in our formal syntax for values.

```
{
person:
  {
   firstname: "John",
   lastname: "Smith",
   coordinates: [10, null, 40]
  }
}
```

Fig. 1 A JSON record r_1.

The type language we adopt is extremely simple (see Section 4). Basic values are abstracted into basic 3.9
data types (String, Number, Boolean and Null), complex values are abstracted by introducing record and array type constructors, and a union type constructor is used to add expressive power to the type language. To illustrate the type language, observe in Figure 2 the schema that corresponds to the record r_1 given in Figure 1: the record structure is exactly described, while the array type lists the possible types of the elements and abstracts from their number and order.

```
{
person:
  {
   firstname: Str,
   lastname: Str,
   coordinates: [Num + Null]
  }
}
```

Fig. 2 A JSON record schema S_1 for r_1.

Our schema inference approach is composed by two phases: individual type inference, that can be implemented by a Map operation, and type reduction, that can be implemented as a Reduce operation.

Individual type inference Individual type inference, during the Map phase, infers a type for each item in the input JSON collection, and yields a set of distinct types to be combined during the Reduce phase.

As we will show, this is a quite simple and fast operation: it consists of a simple traversal of the input values that produces a type with the same shape of the input data, with the only exception of array types. The types of the elements of an array are combined according to the same *type reduction* algorithm that is employed during the second phase. So, instead of inferring a precise type `[Num, Null, Num]` we infer a more succinct version `[ (Num + Null)* ]` describing arrays of arbitrary length where each value is either a number or a null value (we will omit the Kleene star symbol from our syntax).

Type reduction Type reduction is the second step of our approach and consists of iteratively merging the types produced during the Map phase. It is performed during the Reduce phase in a distributed fashion, and it relies on a binary operator that is associative and commutative. Binary reduction is invoked over two types T_1 and T_2, and returns a supertype of the inputs. To do so, types are first compared according to an Equivalence Relation E, that is a parameter of the algorithm. If they are similar enough, according to E, then they are fused, through a synchronized top-down traversal that identifies and combines the common parts, according to the same similarity parameter E. If they are not similar, the result is just the union type $T_1 + T_2$. A coarser relation will merge more pairs, hence producing a type that is more compact but less precise, while a finer relation will have the opposite effect.

The coarsest equivalence relation that we consider is the *kind equivalence*, that fuses two types when they are equal basic types, or are both records, or are both arrays, with no further condition. This yields the most compact result, and reduces types as follows.

- Atomic types: identical atomic types are collapsed, while different types are combined using the union operator.
- Record types:
  - keys without a match in the other record type are just copied to the fused record type and are marked as optional;
  - matching keys from both types are collapsed and their respective types are recursively fused.

As an example, consider the schema S_1 for record r_1 (see Figures 1 and 2), and assume that one wants to merge S_1 with a new schema S_2, shown in Figure 3, describing records that are similar to r_1, but feature a supplementary *email* field, a null *lastname* field, and number only *coordinates*.

```
{
person:
  {
   firstname: Str,
   lastname: Null,
   coordinates: [Num],
   email:  Str
  }
}
```

Fig. 3 A JSON record schema S_2.

With kind equivalence, the schema obtained by merging S_1 and S_2 is the one shown in Figure 4, where the question mark '?' in the *email* field indicates its optionality.

The two schemas are fused since they are of the same kind. Fusion recursively applies for the same reason to the nested record schemas, but a union type is created for *lastname* to capture the two possible types of this field, while *email* is made optional. Furthermore, merging the two array types for *coordinates* yields a union

```
{
person:
  {
   firstname: Str,
   lastname: Str + Null,
   coordinates: [Num + Null],
   email:  Str?
  }
}
```

Fig. 4 Reduction of schemas S_1 and S_2, according to kind-equivalence.

```
{
person:
  {
   firstname: Str,
   lastname: Str,
   coordinates: [Num + Null]
  } +
  {
   firstname: Str,
   lastname: Null,
   coordinates: [Num],
   email:  Str
  }
}
```

Fig. 5 Reduction of schemas S_1 and S_2, according to label equivalence.

of all the distinct types in the body of the two array types.

In the more general case where the array schemas to be fused contain nested record and array types, the fusion process recursively combines the types of the same kind. For instance, the fusion of $[\,\texttt{Str} + \{A{:}\texttt{Str}, B{:}\texttt{Num}\}\,]$ and $[\,\texttt{Num} + \{B{:}\texttt{Str}, C{:}\texttt{Num}\}\,]$ yields

$$[\,\texttt{Str} + \texttt{Num} + \{A{:}\texttt{Str}?, B{:}(\texttt{Str} + \texttt{Num}), C{:}\texttt{Num}?\}\,].$$

A finer equivalence is the *label equivalence* equivalence, which restricts kind equivalence by imposing that only record schemas having the same set of keys are fused. If we go back to the previous example, fusion according to label equivalence returns the schema in Figure 5, which is less succinct but more precise, as, for example, it relates the *Null* type of the last-name with the presence of an *email* key.

It is not possible in general to affirm that one equivalence is better than the other one. For example, the experiments we report in Section 7 show that label equivalence may yield more precise schemas, describing interesting *correlation* properties between different fields, but it generates types that are bigger, and the growth factor is significative.

The precision/size tradeoff can be illustrated even more clearly by the following small example. Consider the fusion of three record schemas, each corresponding to a different input value: $\{A{:}\texttt{Str}, B{:}\texttt{Str}\}$, $\{A{:}\texttt{Str}, B{:}\texttt{Str}\}$, and $\{C{:}\texttt{Str}, D{:}\texttt{Str}\}$. The fused type we obtain with kind equivalence is

$$\{A{:}\texttt{Str}?, B{:}\texttt{Str}?, C{:}\texttt{Str}?, D{:}\texttt{Str}?\}$$

This schema precisely describes what fields may be present, but it is compatible with 2^4 possible different record structures, while we had only two different structures in the input data. Hence, we can observe that as soon as irregularity starts appearing in the data, we may have a big loss of precision in terms of which sets of keys do actually appear. By means of the label equivalence, instead, for the above three record schemas we obtain the more precise, still rather succinct schema

$$\{A{:}\texttt{Str}, B{:}\texttt{Str}\} + \{C{:}\texttt{Str}, D{:}\texttt{Str}\}$$

that specifies, among other information, that A and B fields always co-occur. On the other hand, if we have high heterogeneity of data, and most of the 2^4 subsets of keys do actually appear in the data, then label equivalence would yield a huge schema, while kind equivalence would ensure a good level of succinctness with limited loss of information. Hence, no equivalence is optimal in general, and the higher information content of label equivalence may, or may not, be worth its bigger size, depending on the regularity of the data and, crucially, on the subjective relevance of correlation information for the person who uses the algorithm.

Apart from these two equivalences, two more equivalences will be later presented, but their interest is essentially theoretical, while we believe that kind equivalence and label equivalence can both be quite useful in practice.

3 Related Work

The problem of inferring structural information from JSON data collections has recently gained the attention of the database research community. The closest work to ours is the very preliminary investigation that we presented in [18] and the more complete treatment we presented in [7]. Major additions of the present work wrt [7] have already been outlined in the Introduction (Remark 1). In addition, we recently presented preliminary results that extend this work in the direction of counting types [9], that are types that include frequency indicators describing the number of times values of a given type occur in the input dataset.

In [35], Wang et al. present a framework for efficiently managing a schema repository for JSON document stores. The proposed approach relies on a notion of JSON schema called *skeleton*. In a nutshell, a skeleton is a collection of trees describing structures that frequently appear in the objects of a JSON data collection. In particular, the skeleton may totally miss information about paths that can be traversed in some of the JSON objects. In contrast, our approach enables the creation of a *path-covering* yet succinct schema description of the input JSON dataset. As already said, having such a *path-covering* structural description is of vital importance for many tasks, like query optimisation, defining and enforcing access-control security policies,

and, importantly, giving the user a global structural vision of the database that can help her in querying and exploring the data in an effective way. Another important application of *path-covering* schema information is query type checking: as illustrated in [18], our inferred schemas can be used to make type checking of Pig Latin scripts much stronger.

In [31], motivated by the need of laying the formal foundations for the JSON Schema language [5], Pezoa et al. present the formal semantics of that language, as well as a theoretical study of its expressive power and validation problem. Along the lines of [31], Bourhis et al. [14] have recently laid the foundations for a logical characterization and formal study for both schema and query languages for JSON. While these works do not deal with the schema inference problem, our schema language can be seen as a core part of the JSON Schema language studied in these works, and shares union types and repetition types with them. These constructors are at the basis of our technique to collapse several schemas into a more succinct one. An alternative proposal for typing JSON data is JSound [4], whose language is quite restrictive with respect to ours and JSON Schema, and specifically it lacks union types.

In [19], Abadi and DiScala deal with the problem of automatically transforming denormalised, nested JSON data into normalised relational data that can be stored into a RDBMS; this is achieved by means of a schema generation algorithm that learns the *normalised, relational* schema from data. Differently from that work, we deal with schemas that are far from being relational, and are closer to tree grammars [28]. Furthermore, the approach proposed in [19] ignores the original structure of the JSON input dataset and, instead, depends on patterns in the attribute data values (functional dependencies) to guide its schema generation. So, that approach is complementary to ours.

In [25], Liu et al. propose storage, querying, and indexing principles enabling RDBMSs to manage JSON. The paper does not deal with schema inference, but indicates a possible optimisation of their framework based on the identification of common attributes in JSON objects that can be captured by a relational schema for optimization purposes. In [32], Scherzinger et al. propose a plugin to track changes in object-NoSQL mappings. The technique is currently limited to only detect mismatches between base types (e.g., Boolean, Integer, String), and the authors claim that a wider knowledge of schema information is needed to enable the detection of other kinds of changes, like, for instance, the removal or renaming of attributes.

In a recent work [24], Li et al. present streaming techniques for efficiently parsing and importing JSON data for analytics tasks, as well as a JSON parser based on those techniques (i.e., Mison). While this approach does not primarily deal with schema inference, it infers structural information of data on the fly in order to detect and prune parts of the data that are not needed by a given analytics task. Our approach can be considered as complementary to that described in [24], since it could be used to improve the effectiveness of Mison.

In [13], Bonetta and Brantner present FAD.JS, a *speculative*, JIT-based JSON encoder and decoder designed for the Oracle Graal.js JavaScript runtime. It exploits data access patterns to optimize both encoding and decoding: indeed, FAD.JS relies on the assumption that most applications never use all the fields of input objects, and, for instance, skips unneeded object fields during JSON object parsing. As for Mison, our schema inference approach can be considered as complementary to FAD.JS.

In the context of NoSQL systems (e.g. MongoDB), recent efforts have been dedicated to the problem of implementing tools for JSON schema inference. A JavaScript library for JSON, called mongodb-schema, is presented in [33]. This tool analyzes JSON objects pulled from MongoDB, and processes them in a streaming fashion. Due to the lack of any formal specification and experimental evaluation, we directly inspected the mongodb-schema source code, and figured out that the tool associates types to paths found in JSON objects, and returns schemas that seem to be equivalent to those produced by our approach under the kind equivalence. Differently from our approach, this system processes data in a centralized fashion, it does not provide a parametric approach, and, in particular, it is not able to infer information describing field correlation. Studio 3T [23] is a commercial front-end for MongoDB that offers a very simple schema inference and analysis feature, but it is not able to merge similar types, and the resulting schemas can have a huge size, which is comparable to that of the input data. In [34], a python-based tool is described, called Skinfer, which infers JSON Schemas from a collection of JSON objects. Skinfer exploits two different functions for inferring a schema from an object and for merging two schemas; schema merging is limited to record types only, and cannot be recursively applied to objects nested inside arrays.

In the context of Spark, the Spark Dataframe schema extraction [6] is a very interesting tool for the automated extraction of a schema from JSON datasets. To the best of our knowledge, Spark is the only tool that supports a distributed schema inference technique for input data; other systems, like Jaql [12], exploit schema information for inferring the output schema of a query, but still require an externally supplied schema for input

data, and perform output schema inference only locally on a single machine. This technique is, however, less precise than ours, since the type language lacks union types, and the inference algorithm resorts to `Str` on strongly heterogeneous collections of data. For instance, an array `[17, {name: "Bear"}]` is typed as an array of strings, as we observed by inspecting the results we got on our datasets (described in Section 7).

Another major difference of this technique wrt our approach is the absence of parametricity, which would make little sense in their approach, given the limited expressivity of the type language. However, the system has excellent scalability properties and is quite efficient, and, for this reason, we choose it as a yardstick for our performance experiments (Section 7).

It is important to state that the problem of schema inference has already been addressed in the past in the context of semi-structured and XML data models. A typical XML data summary is in the form of a *dataguide* [22], which abstracts away from important properties that our schemas describe, like optional/mandatory fields and correlation between them. In [29] and [30], Nestorov et al. describe an approach to extract a schema from semistructured data. They propose an object-oriented type system where nodes are captured by classes built starting from nodes sharing the same incoming and outgoing edges and where data edges are generalized to relations between the classes. In [30], the problem of building a type out of a collection of semistructured documents is studied. The emphasis is put on minimizing the size of the resulting type while maximizing its precision. Although that work considers a very general data model captured by graphs, it does not suit our context. Firstly, we consider the JSON model, that is tree-shaped and that features specific constructs such as arrays that are not captured by the semi-structured data model. Secondly, we aim at processing potentially large datasets efficiently, a problem that is not directly addressed in [29] and [30]. On the other side, these papers address the important problem of an optimal aggregation of different record types that are similar in structure. We do not face this problem at all, since we leave the choice of the equivalence relation to the user, and we only work on the extreme cases of either performing a full aggregation of all record types or keeping them separated as soon one key is different. In this sense, our work is orthogonal to theirs.

More recent efforts on XML schema inference (see [20] and works cited therein) are also worth mentioning since they are somewhat related to our approach. The aim of these approaches is to infer restricted, yet expressive enough forms of regular expressions starting from a positive set of strings representing element contexts of XML documents. While XML and JSON both allow one to represent tree-shaped data, they have radical differences that make existing XML related approaches difficult to apply to the JSON setting. Namely, since a regular language is the most natural tool to describe the content model for an XML element type, inference of regular expressions is a central issue in the XML schema inference problem, while the same tool has a limited utility in the field of JSON schema inference, where a central role is played by the distinction between array types and record types, that are both represented as element types in the XML world.

Still in the context of XML schema inference, techniques like [11] are based on the construction of an NFA from strings, and on its transformation in order to build regular expression validating input strings corresponding to XML element content. In our framework, the coexistence of (possibly nested) records and arrays with their specific features (uniqueness of record keys, possible heterogeneity in the array content, etc.) make the adoption of NFAs very difficult, even without considering distribution.

In [17] Ciucanu and Staworko discuss schema inference for unordered XML. Since they ignore order, they do not rely on NFAs but rather on label-to-label relationships: which labels appear below a given label, and which pairs of labels appear together in the same element. In their work they study the *learnability* problem: given a class of languages $\mathcal{C}$, does a PTime algorithm exists that, given a set of words D, infers a minimal language in $\mathcal{C}$ that includes D? They study this problem for two classes of languages, MS and DMS, and they consider two situations where only positive samples are given, and where both positive and negative samples are given. They give interesting results on the theme of language learnability, a theme that we decided to ignore since the learnability of the schema language that we study is quite obvious. However, they are not interested in the Big Data setting, hence they work on PTime algorithms designed for sequential execution, rather than looking for linear and parallelizable algorithms as we do. The expressive power of the language they consider is considerably lower than ours, since they require that no key appears twice in the schema.

In [26], Lohrey et al. describe compression techniques of XML trees under the assumption that the data collection is either ordered (document-centric) or unordered (data-centric). As in most compression techniques, their approaches identifies common structural information in the input data, and share with our approach the idea of merging sub-trees. However, Lohrey et al. focus on giving lower-bounds in terms of the size of compressed trees under the assumption that XML

trees are unordered, and postpone the study of compression of JSON data as future work, as JSON data have, at once, ordered (arrays) and unordered components (records). Furthermore, the main focus in [26] is on centralized techniques for data compression only, while our interest here is on schemas, that are usually characterized by an abstraction level which is much higher than that of compression, especially when the 4.9
chosen type equivalence is kind-based.

In [16] Cebiric et al. study techniques for creating summaries for RDF data. These summaries are essentially DataGuides for RDF data and it is far from clear how their technique can be adapted to JSON data. To conclude, we would like to stress that none of these XML/RDF related approaches is designed to deal with massive datasets, which is one of the central points in 4.9
our work.

4 Data Model and Type Language

JSON values are either basic values, records, or arrays. Basic values B include the null value, booleans, numbers n, and strings s. Records represent sets of fields, each field being a key-value pair $(l, \mathcal{V})$, and arrays represent sequences of values. We will use J to range over JSON expressions and $\mathcal{V}$ to range over the values denoted by such expressions, according to the semantics defined below, but the two notions are so similar that we will often ignore this distinction.

We will only consider here records without repeated keys. Extending our approach to the case of records with repeated keys is very simple, but we did not yet find any practical example where this extension were 1.8
useful. We use $Keys(R)$ to denote the set of the keys of a record R. In JSON syntax, a key is itself a string, hence is surrounded by quotes; we avoid these quotes in our notation, that is, we write { name : "John" } rather than { "name" : "John" }.

Notation 1 *$Sets(S)$ is the set of all subsets of S.*
$FSets(S)$ is the set of all finite subsets of S.
$Lists(S)$ is the set of all finite lists whose elements are in S.
To reduce the confusion between JSON values and mathematical objects, we denote a set as $\{\!|\, a_1, \ldots, a_n \,|\!\}$ and a list as $\langle\!\langle a_1, \ldots, a_n \rangle\!\rangle$.

Syntax

$J ::= B \mid R \mid A$	**JSON expressions**
$B ::=$ null $\mid$ true $\mid$ false $\mid n \mid s$ $n \in$ **Number**, $s \in$ **String**	**Basic values**
$R ::= \{l_1 : J_1, \ldots, l_n : J_n\}$ $n \geq 0, \quad i \neq j \Rightarrow l_i \neq l_j$	**Records**
$A ::= [J_1, \ldots, J_n] \quad n \geq 0$	**Arrays**

Semantics of JSON expressions is described below. For each kind of values we also report the *domain*, that is, the set that contains the denotation of the corresponding JSON values. For instance, *BasicValues* includes numbers, strings, etc., used to interpret JSON basic values, while the semantics of a JSON record is in the domain $FSets(Keys \times Values)$, where $Keys$ is the infinite set of all possible record keys, and $Values$ is the set of all possible denotations of JSON values produced by the semantic function $[\![\,\_\,]\!]$. 3.10-5

Semantics

Basic Values
$Domain : BasicValues$
$[\![\, B \,]\!] \triangleq B$

Records
$Domain : FSets(Keys \times Values)$
$[\![\, \{l_1 : J_1, \ldots, l_n : J_n\} \,]\!] \triangleq \{\!|\, (l_1, [\![\, J_1 \,]\!]), \ldots, (l_n, [\![\, J_n \,]\!]) \,|\!\}$

Arrays
$Domain : Lists(Values)$
$[\![\, [J_1, \ldots, J_n] \,]\!] \triangleq \langle\!\langle [\![\, J_1 \,]\!], \ldots, [\![\, J_n \,]\!] \rangle\!\rangle$

Types in our type system obey the following grammar. 3.9,3.10-1

Syntax of types

$\mathcal{T} ::= \emptyset \mid \mathcal{T} + \mathcal{T} \mid \mathcal{S}$	**Types**
$\mathcal{S} ::= \mathcal{B} \mid \mathcal{R} \mid \mathcal{A}$	**Struct. types**
$\mathcal{B} ::=$ `Null` $\mid$ `Bool` $\mid$ `Num` $\mid$ `Str`	**Basic types**
$\mathcal{R} ::= \{l_1 : \mathcal{T}_1 \mathtt{q}_1, \ldots, l_n : \mathcal{T}_n \mathtt{q}_n\} \; n \geq 0$	**Record types**
$\mathtt{q} ::= ! \mid ?$	**Quantifiers**
$\mathcal{A} ::= [\mathcal{T}]$	**Array types**

A type describes a, possibly infinite, set of JSON values. The empty type $\emptyset$ contains no value and is 3.10-2
the neutral element for the union operator $\_ + \_$, which is associative and commutative. Hence, any type $\mathcal{T}$ is either equivalent to $\emptyset$ or to a finite union $\mathcal{S}_1 + \ldots + \mathcal{S}_n$ of *structural types* $\mathcal{S}_i$, where a structural type is a type that describes either a set of basic values ($\mathcal{B}$), a set of records ($\mathcal{R}$) or a set of arrays ($\mathcal{A}$). Basic types are standard. A record is a set of pairs $(l, \mathcal{V})$, and a record type specifies which keys must appear or may appear in a record, and the associated types. For example, a type $\{l$: `Num`?, m : (`Str` + `Null`)!$\}$ describes records where l is optional and, if present, contains a number, while the m field is mandatory and may contain either null or a string, so that both $\{\!|\, (l, 3), (m, \text{null}) \,|\!\}$ and $\{\!|\, (m, \text{"}abc\text{"}) \,|\!\}$ belong to $[\![\, \{l$: `Num`?, m : (`Str`+`Null`)!$\} \,]\!]$; the semantics of this record type, hence, is a set of finite sets of pairs. In our examples we will often omit the '!'

for mandatory fields and only annotate optional fields with '?'.

Array types specify the type of elements that may appear in the corresponding arrays. For example, a type $[\,\mathtt{Str}+\{\mathit{name}\colon \mathtt{Str}\}\,]$ describes arrays that may contain any number of values $\langle\!\langle \mathcal{V}_1, \ldots, \mathcal{V}_n \rangle\!\rangle$ where each $\mathcal{V}_i$ is either a string or a record with a *name* key associated to a string. By this definition, the type $[\,\emptyset\,]$ contains arrays that contain any number of values that belong to the empty type; since no value belongs to $\emptyset$, the only possible value for $[\,\emptyset\,]$ is the empty array.

Observe that types and structural types are defined by mutual recursion: a type is the union of a set of structural types and, vice versa, the content of a structural type (the fields of a record type and the content type of an array type) is defined in terms of types. 3.4

Semantics of types is formally specified as follows.

Basic types

$\mathit{Domain} : \mathit{Sets}(\mathit{BasicValues})$

$[\![\,\mathtt{Null}\,]\!] \triangleq \{\!|\ \text{null}\ |\!\}$

$[\![\,\mathtt{Bool}\,]\!] \triangleq \{\!|\ \text{true},\ \text{false}\ |\!\}$

$[\![\,\mathtt{Num}\,]\!] \triangleq \mathbf{Number}$

$[\![\,\mathtt{Str}\,]\!] \triangleq \mathbf{String}$

Record types

$\mathit{Domain} : \mathit{Sets}(\mathit{FSets}(\mathit{Keys} \times \mathit{Values}))$

$[\![\,\{\,\}\,]\!] \triangleq \{\!|\ \emptyset\ |\!\}$

$[\![\,\{l : \mathcal{T}!\}\,]\!] \triangleq \{\!|\ \{\!|\ (l, \mathcal{V})\ |\!\} \mid \mathcal{V} \in [\![\,\mathcal{T}\,]\!]\ |\!\}$

$[\![\,\{l : \mathcal{T}?\}\,]\!] \triangleq [\![\,\{l : \mathcal{T}!\}\,]\!] \cup [\![\,\{\,\}\,]\!]$

$$[\![\,\{l_1 : \mathcal{T}_1\mathtt{q}_1, \ldots, l_n : \mathcal{T}_n\mathtt{q}_n\}\,]\!] \triangleq \{\!|\ R_1 \cup \ldots \cup R_n \mid R_1 \in [\![\,\{l_1 : \mathcal{T}_1\mathtt{q}_1\}\,]\!], \ldots, R_n \in [\![\,\{l_n : \mathcal{T}_n\mathtt{q}_n\}\,]\!]\ |\!\}$$

Array types

$\mathit{Domain} : \mathit{Sets}(\mathit{Lists}(\mathit{Values}))$

$[\![\,[\mathcal{T}]\,]\!] \triangleq \{\!|\ \langle\!\langle \mathcal{V}_1, \ldots, \mathcal{V}_n \rangle\!\rangle \mid n \geq 0,\ \mathcal{V}_i \in [\![\,\mathcal{T}\,]\!]\ |\!\}$

Union types

$[\![\,\emptyset\,]\!] \triangleq \emptyset$

$[\![\,\mathcal{T} + \mathcal{U}\,]\!] \triangleq [\![\,\mathcal{T}\,]\!] \cup [\![\,\mathcal{U}\,]\!]$

The basic idea behind the *type reduction* mechanism that we are going to present is to merge types that have the same *kind*, that is, records with records, arrays with arrays, numbers with numbers, and so on, provided that they are "similar enough", as discussed later. The following *kind*() function, that maps each structural type to its outermost constructor, is used to formalize the notion of 'types having the same kind'.

$$\mathit{kind}(\mathtt{Null}) = \mathtt{Null} \qquad \mathit{kind}(\mathtt{Bool}) = \mathtt{Bool}$$
$$\mathit{kind}(\mathtt{Num}) = \mathtt{Num} \qquad \mathit{kind}(\mathtt{Str}) = \mathtt{Str}$$
$$\mathit{kind}(\mathcal{R}) = \{\,\} \qquad \mathit{kind}(\mathcal{A}) = [\,]$$

Later on, in order to express the correctness of the fusion process we rely on the usual notions of sub-typing (type inclusion) and type equivalence, that we define here.

Definition 1 ($\mathcal{T} \leq \mathcal{U}$, $\mathcal{T} \simeq \mathcal{U}$) Let $\mathcal{T}$ and $\mathcal{U}$ be two types.

- $\mathcal{T}$ is a sub-type of $\mathcal{U}$, written as $\mathcal{T} \leq \mathcal{U}$, if and only if $[\![\,\mathcal{T}\,]\!] \subseteq [\![\,\mathcal{U}\,]\!]$;
- $\mathcal{T}$ is equivalent to $\mathcal{U}$, written as $\mathcal{T} \simeq \mathcal{U}$, if and only if $[\![\,\mathcal{T}\,]\!] = [\![\,\mathcal{U}\,]\!]$.

It is useful to ignore the order of fields in record types, to consider union types modulo associativity and commutativity of '+', modulo repetition of the united types, and to ignore $\emptyset$ inside a union. We formalize this abstract view of types using the following equivalence $\mathcal{T}_1 \doteq \mathcal{T}_2$.

Definition 2 ($\mathcal{T} \doteq \mathcal{U}$) We say that two types are syntactically congruent ($\doteq$) when one can be transformed into the other by repeated application of the following rules, that can be applied in any direction and anywhere inside the type. 4.2

Syntactic congruence rules

$$\mathcal{T}_1 + \mathcal{T}_2 \doteq \mathcal{T}_2 + \mathcal{T}_1$$
$$\mathcal{T}_1 + (\mathcal{T}_2 + \mathcal{T}_3) \doteq (\mathcal{T}_1 + \mathcal{T}_2) + \mathcal{T}_3$$
$$\mathcal{T} + \emptyset \doteq \mathcal{T}$$
$$\mathcal{T} + \mathcal{T} \doteq \mathcal{T}$$
$$\{l_1 : \mathcal{T}_1\mathtt{q}_1, \ldots, l_k : \mathcal{T}_k\mathtt{q}_k\} \doteq \{l_{\sigma(1)} : \mathcal{T}_{\sigma(1)}\mathtt{q}_{\sigma(1)}, \ldots, l_{\sigma(k)} : \mathcal{T}_{\sigma(k)}\mathtt{q}_{\sigma(k)}\} \text{ with } \sigma \text{ bijective on } \{1, \ldots, k\}$$

Hence, two types $\mathcal{T}_1$ and $\mathcal{T}_2$ are syntactically congruent if their congruence can be proved by reordering the components of union types and record types, disregarding duplicates and the presence of $\emptyset$ in a union type. 4.2

Types that are syntactically congruent are equivalent, that is, they have the same semantics. 3.13

Property 1 For any $\mathcal{T}_1$ and $\mathcal{T}_2$: $\mathcal{T}_1 \doteq \mathcal{T}_2 \;\Rightarrow\; \mathcal{T}_1 \simeq \mathcal{T}_2$

Proof All the five rules of Definition 2 transform a type into a type that has the same semantics. For the first four rules, this follows immediately from the fact that $[\![\,\mathcal{T}_1 + \mathcal{T}_2\,]\!] = [\![\,\mathcal{T}_1\,]\!] \cup [\![\,\mathcal{T}_2\,]\!]$. For the last rule, it descends from the fact that the order of labels is irrelevant in the definition of the semantics of record types. 3.13

Observe that the converse does not hold. For example, we have that 3.12

$$[\,\mathtt{Num} + \mathtt{Str}\,] + [\,\mathtt{Str}\,] \;\simeq\; [\,\mathtt{Num} + \mathtt{Str}\,]$$

since [Str] denotes a subset of [Num + Str], but the two types are not syntactically congruent.

In the sequel, we will need to extract all structural addends out of a union type, and to transform a set of structural types into their union. To this aim we define a pair of operators, $\circ\mathcal{T}$ for the extraction and $\oplus\mathcal{M}$ to rebuild the original type, such that, for example:

$$\circ((\mathtt{Num} + \mathtt{Str}) + (\mathtt{Num} + \emptyset)) = \{\!\!\{\ \mathtt{Num}, \mathtt{Str}\ \}\!\!\}$$

and

$$\oplus(\{\!\!\{\ \mathtt{Num}, \mathtt{Str}\ \}\!\!\}) \doteq \mathtt{Num} + \mathtt{Str}.$$

3.9

Coherently with our notion of syntactic congruence, the operator $\circ(\mathcal{T})$ ignores duplicates, empty types, and 3.9 the order of the addends. It is easy to see that

$$\oplus(\circ(\mathcal{T})) \doteq \mathcal{T}.$$

Definition 3 ($\circ\mathcal{T}$ (addends of $\mathcal{T}$), $\oplus\mathcal{M}$) For any type $\mathcal{T}$ and for any set $\mathcal{M}$ of structural types, the operators $\circ\mathcal{T}$ and $\oplus\mathcal{M}$ are defined as follows. The elements of $\circ\mathcal{T}$ are called the *addends* of $\mathcal{T}$.

$$\begin{array}{lcll}
\circ(\mathcal{T}_1 + \mathcal{T}_2) & \triangleq & \circ\mathcal{T}_1 \cup \circ\mathcal{T}_2 & \\
\circ\emptyset & \triangleq & \emptyset & \\
\circ\mathcal{S} & \triangleq & \{\!\!\{\ \mathcal{S}\ \}\!\!\} & \\
\oplus(\emptyset) & \triangleq & \emptyset & \\
\oplus(\{\!\!\{\ \mathcal{S}\ \}\!\!\}) & \triangleq & \mathcal{S} & \\
\oplus(\{\!\!\{\ \mathcal{S}\ \}\!\!\} \cup \mathcal{M}) & \triangleq & \mathcal{S} + (\oplus\mathcal{M}) & \text{if } M \neq \emptyset
\end{array}$$

We then define a similar pair of operators for the record types: $\diamond\mathcal{R}$ to extract the key-type-quantifier triples out of $\mathcal{R}$, and $\underline{\{}\ \mathcal{M}\ \underline{\}}$ to rebuild the original type, so that $\underline{\{}\diamond\mathcal{R}\underline{\}}$ is equal to $\mathcal{R}$, modulo the field order.

Definition 4 ($\diamond\mathcal{R}$, $\underline{\{}\ S\ \underline{\}}$)

$$\diamond\{l_1 : \mathcal{T}_1\mathtt{q}_1, \ldots, l_k : \mathcal{T}_k\mathtt{q}_k\} \triangleq \{\!\!\{\ (l_1, \mathcal{T}_1, \mathtt{q}_1), \ldots, (l_k, \mathcal{T}_k, \mathtt{q}_k)\ \}\!\!\}$$

$$\underline{\{}\ \{\!\!\{\ (l_1, \mathcal{T}_1, \mathtt{q}_1), \ldots, (l_k, \mathcal{T}_k, \mathtt{q}_k)\ \}\!\!\}\ \underline{\}} \triangleq \{l_1 : \mathcal{T}_1\mathtt{q}_1, \ldots, l_k : \mathcal{T}_k\mathtt{q}_k\}$$

5 Type Reduction

5.1 Parametric reduction

Our type-inference approach infers a type for each JSON value and then merges all the inferred types using a parametric *Reduce* operator, that is also used to merge the types of the elements of each array found in a JSON 3.1 value. Reduction constitutes the core of our approach.

The reduction process is sound as long as the merged type is a supertype of its arguments, a property that is not very restrictive, and can be satisfied by many different definitions of the *Reduce* operator. As will be soon exemplified, each different definition gives rise to a different trade-off between precision and size of the produced type. The fundamental feature of our approach 3.1 is its parametricity, that is, the fact that it can be tuned in order to reach a different precision-size trade-off. This tuning is obtained by providing a parametric definition for the *Reduce* operator. 4.4

In a nutshell, the E parameter of $Reduce(\mathcal{T}_1, \mathcal{T}_2, E)$ is an equivalence relation that governs its behavior: two types are *fused* when they are E-equivalent, while they are kept separated when they are not, so that the E parameter provides an abstract specification of the precision of our type system.

In the next section we define the notions of *reduction* and *fusion* and we exemplify some possible choices for the E parameter.

5.2 Reduction and fusion

1.0,1.15

We introduce here the terms *reduction* and *fusion*, that will be used with a precise technical meaning.

We use *reduction* to indicate the process of finding a common supertype of two (or more) types $\mathcal{T}_1$ and $\mathcal{T}_2$, which may be as big and precise as $\mathcal{T}_1 + \mathcal{T}_2$ or may be more compact. We use *fusion* to indicate the process of finding a common *structural* supertype of two (or more) *structural* types $\mathcal{S}_1$ and $\mathcal{S}_2$. Reduction of two types such as $(\mathcal{S}_1 + \mathcal{S}_2 + \mathcal{S}_3)$ and $(\mathcal{S}_4 + \mathcal{S}_5)$ is based on the fusion of their structural addends, and fusion of two structural types such as $[\mathcal{T}_1]$ and $[\mathcal{T}_2]$ is based on the reduction of their internal types (and similarly for record types). Reduction and fusion are, in general, not uniquely defined, as we have discussed many times.

While every two types $\mathcal{T}_1$ and $\mathcal{T}_2$ always admit the trivial *reduction* $\mathcal{T}_1 + \mathcal{T}_2$, the type $\mathcal{S}_1 + \mathcal{S}_2$ is not a *fusion* of $\mathcal{S}_1$ and $\mathcal{S}_2$, since it is not structural, because of the outermost $+$. It is easy to see that $\mathcal{S}_1 + \mathcal{S}_2 \leq \mathcal{S}_3$ implies that $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ all have the same kind, since no value belongs to two types of different kinds, hence $\mathcal{S}_1$ and $\mathcal{S}_3$ must have the same kind, and the same holds for $\mathcal{S}_2$ and $\mathcal{S}_3$.[1] It follows that $\mathcal{S}_1$ and $\mathcal{S}_2$ can only be fused if they have the same kind.

We now briefly discuss how structural types with the same kind can be fused.

[1] We are here ignoring empty structural types, which are record types where one mandatory field has type $\emptyset$, since they are never inferred, and we could even forbid them in the syntax.

Fusing base types is easy, recalling that any base type has its own kind: a union type $\texttt{Num} + \texttt{Num}$ can be rewritten as `Num`, obtaining size reduction with no loss of information, and the same holds for `Str`, `Bool` and `Null`. Two different record types can be fused by exploiting the presence of optional fields, so that

$$\{a{:}\texttt{Num}, b{:}\texttt{Num}\} + \{b{:}\texttt{Num}, c{:}\texttt{Num}\}$$

can be rewritten as

$$\{a{:}\texttt{Num}?, b{:}\texttt{Num}, c{:}\texttt{Num}?\}$$

that is, the types of the shared keys are combined and reduced, and keys that are not shared are marked as optional, which makes the final type more compact but (possibly) less precise, as in this case. Two array types can also be fused, by rewriting $[\mathcal{T}_1] + [\mathcal{T}_2]$ into its supertype $[\mathcal{T}_1 + \mathcal{T}_2]$ and by recursively reducing the internal type $\mathcal{T}_1 + \mathcal{T}_2$.

Hence, when two structural types have the same kind, they always admit a structural fusion. However, such fusion may yield a loss of type information. For example, if we rewrite

$$\{a{:}\texttt{Num}, b{:}\texttt{Num}, c{:}\texttt{Num}\} + \{a{:}\texttt{Num}, b{:}\texttt{Num}\}$$

into

$$\{a{:}\texttt{Num}, b{:}\texttt{Num}, c{:}\texttt{Num}?\}$$

there is no information loss, as the new type, despite being shorter, has the same semantics as the union type. However, assume we rewrite

$$\{a{:}\texttt{Num}, b{:}\texttt{Num}\} + \{b{:}\texttt{Str}, c{:}\texttt{Str}, d{:}\texttt{Str}\}$$

as

$$\{a{:}\texttt{Num}?, b{:}(\texttt{Num} + \texttt{Str}), c{:}\texttt{Str}?, d{:}\texttt{Str}?\}.$$

The original type specifies that keys a and c are mutually exclusive, while c and d always appear together. The original type specifies that b is a number if, and only if, the a key is present. The reduced type correctly reports which fields are always, or sometimes, present, and the corresponding type information, but has lost all the correlation information between the presence, and the types, of the different fields.

The same is true for array types. Assume we rewrite $[\mathcal{T}_1] + [\mathcal{T}_2]$ into its supertype $[\mathcal{T}_1 + \mathcal{T}_2]$: while the original type describes arrays that are either uniformly composed by elements of $\mathcal{T}_1$ only, or by elements of $\mathcal{T}_2$ only, the fused type also admits arrays with a mixed content.

While in some situations the information loss is perfectly acceptable, in the same way as, at the type level, it is acceptable to abstract all different numbers into just `Num`, in other cases the field correlation information is quite important. There exists no "optimum trade-off" between compactness and precision, since it depends on the precision requirements of the specific task that the programmer, or the data analyst, needs to perform.

Hence, we define a reduction operation that is parametrized over an equivalence relation (ER) E, and which fuses two structural types if and only if they are E-equivalent. In this way, a finer E will fuse less pairs, yielding a result that is bigger but more informative. A coarser equivalence will give a different tradeoff, since it will fuse more pairs, hence producing a result that is more compact but less informative. In the extreme cases, the identity ER will only fuse identical types, yielding a huge type with no information loss, while a relation that relates every two types with the same kind will return a much smaller type, with a higher information loss.

5.3 Formal definition of reduction and fusion

We need a couple of preliminary definitions to formalize the reduction operation. We first define the notion of a kind-respecting ER.

Definition 5 (Kind-respecting ER, $\mathcal{K}$ equivalence) A Kind-respecting ER (KER) is an equivalence relation on structural types E such that

$$E(\mathcal{S}_1, \mathcal{S}_2) \;\Rightarrow\; kind(\mathcal{S}_1) = kind(\mathcal{S}_2)$$

$\mathcal{K}(\mathcal{S}_1, \mathcal{S}_2)$ is the maximal KER, defined by

$$\mathcal{K}(\mathcal{S}_1, \mathcal{S}_2) \;\Leftrightarrow\; kind(\mathcal{S}_1) = kind(\mathcal{S}_2)$$

We must now give a name to the basic invariant of our algorithm. Given a KER E, whenever two structural E-equivalent types are operands of a union, our algorithm will fuse them, hence producing a union where no pair of distinct addends are E-equivalent. We say that such a type is E-reduced, as defined below (since any structural type is a type, the definition below applies to structural types as well).

Definition 6 (E-reduced) Given any partial equivalence relation E defined on structural types, a type $\mathcal{T}$ is *E-reduced* iff, for any union type $\mathcal{T}_1$ that is found at any nesting level inside $\mathcal{T}$, no two distinct addends in $\circ\mathcal{T}_1$ are E-equivalent. A set $\mathcal{M}$ of structural types is E-reduced iff the type $\oplus\mathcal{M}$ is E-reduced.

Example 1 Consider the following KERs:

- syntactic congruence $\doteq$;
- kind equivalence $\mathcal{K}$.

Consider the following three types.

$$[\texttt{Num} + \texttt{Str}] + \texttt{Str} + [\texttt{Str} + \texttt{Num}] + \texttt{Str} + [\texttt{Num} + \texttt{Bool}]$$
$$[\texttt{Num} + \texttt{Str}] + \texttt{Str} + [\texttt{Num} + \texttt{Bool}]$$
$$[\texttt{Num} + \texttt{Str} + \texttt{Bool}] + \texttt{Str}$$

The first type is not E-reduced, for any E, since `Str` and [`Num` + `Str`] are repeated. The second type is syntactically congruent to the first (that is, is $\doteq$-equivalent) and is $\doteq$-reduced, but is not $\mathcal{K}$-reduced. Finally, the third type is $\mathcal{K}$-reduced (and hence, a fortiori, $\doteq$-reduced), is more compact than the first two types, but is not semantically equivalent to them, since it is a strict supertype.

We can finally define our $\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)$ operator.

Definition 7 **($\mathtt{q} \cdot \mathtt{q}'$)** Quantifier conjunction $\mathtt{q} \cdot \mathtt{q}'$ is defined as follows.

$$\mathtt{q} \cdot \mathtt{q}' = \begin{cases} ! & \text{if } \mathtt{q} =! \text{ and } \mathtt{q}' =! \\ ? & \text{otherwise} \end{cases}$$

Definition 8 **($\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)$)** For any KER E defined on structural types, for any E-reduced types $\mathcal{T}_1$ and $\mathcal{T}_2$, for any E-reduced structural types $\mathcal{S}_1$ and $\mathcal{S}_2$ having the same kind, the operators $\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)$ and $\mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E)$, are defined as follows, by mutual induction and by cases on the common kind of $\mathcal{S}_1$ and $\mathcal{S}_2$.

$$\begin{aligned}
&\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E) \triangleq \\
&\oplus(\{\!\{\ \mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E) \mid \mathcal{S}_1 \in \circ\mathcal{T}_1,\ \mathcal{S}_2 \in \circ\mathcal{T}_2,\ E(\mathcal{S}_1, \mathcal{S}_2)\ \}\!\} \\
&\quad \cup \{\!\{\ \mathcal{S}_1 \mid \mathcal{S}_1 \in \circ\mathcal{T}_1,\ \nexists \mathcal{S}_2 \in \circ\mathcal{T}_2.\ E(\mathcal{S}_1, \mathcal{S}_2)\ \}\!\} \\
&\quad \cup \{\!\{\ \mathcal{S}_2 \mid \mathcal{S}_2 \in \circ\mathcal{T}_2,\ \nexists \mathcal{S}_1 \in \circ\mathcal{T}_1.\ E(\mathcal{S}_1, \mathcal{S}_2)\ \}\!\} \quad)
\end{aligned}$$

$$\mathit{Fuse}(\mathcal{B}, \mathcal{B}, E) \triangleq \mathcal{B}$$

$$\begin{aligned}
&\mathit{Fuse}(\mathcal{R}_1, \mathcal{R}_2, E) \triangleq \\
&\{\ \{\!\{\ (l, \mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E), \mathtt{q}_1 \cdot \mathtt{q}_2) \\
&\qquad\qquad \mid (l, \mathcal{T}_1, \mathtt{q}_1) \in \diamond\mathcal{R}_1,\ (l, \mathcal{T}_2, \mathtt{q}_2) \in \diamond\mathcal{R}_2\ \}\!\} \\
&\quad \cup \{\!\{\ (l, \mathcal{T}_1, \mathtt{q}_1) \mid (l, \mathcal{T}_1, \mathtt{q}_1) \in \diamond\mathcal{R}_1, \\
&\qquad\qquad \nexists \mathcal{T}_2, \mathtt{q}_2.\ (l, \mathcal{T}_2, \mathtt{q}_2) \in \diamond\mathcal{R}_2\ \}\!\} \\
&\quad \cup \{\!\{\ (l, \mathcal{T}_2, \mathtt{q}_2) \mid (l, \mathcal{T}_2, \mathtt{q}_2) \in \diamond\mathcal{R}_2, \\
&\qquad\qquad \nexists \mathcal{T}_1, \mathtt{q}_1.\ (l, \mathcal{T}_1, \mathtt{q}_1) \in \diamond\mathcal{R}_1\ \}\!\} \\
&\}
\end{aligned}$$

$$\mathit{Fuse}([\mathcal{T}_1], [\mathcal{T}_2], E) \triangleq [\,\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)\,]$$

Reduction of $\mathcal{T}_1$ and $\mathcal{T}_2$ is performed by fusing each addend $\mathcal{S}_1$ of $\mathcal{T}_1$ with any addend $\mathcal{S}_2$ of $\mathcal{T}_2$ that is E-equivalent to $\mathcal{S}_1$. Observe that such equivalent addend $\mathcal{S}_2$, if it exists, is unique: if we had two distinct addends $\mathcal{S}_2$ and $\mathcal{S}_2'$ of $\mathcal{T}_2$ that are both E-equivalent to $\mathcal{S}_1$, they would be E-equivalent as well, by transitivity, and this would contradict the hypothesis that $\mathcal{S}_2$ is E-reduced.

Observe that any addend of $\mathcal{T}_1$ is either fused with an E-equivalent addend from $\mathcal{T}_2$, if it exists, or it is put in the merged type as it is, if no equivalent addend in found in $\mathcal{T}_2$, and the same for the addends of $\mathcal{T}_2$. Hence, each addend contributes once to the reduced type. In the same way, when two record types $\mathcal{R}_1$ and $\mathcal{R}_2$ are fused, every field of each record contributes once to the fused record type.

Our definition of fusion is commutative (modulo syntactic congruence $\doteq$) by construction, and is associative (Theorem 4) when the equivalence relation enjoys the following property, that we call 'stability': the E-fusion of two E-equivalent structural types yields a third type that is still E-equivalent to the original types.

Definition 9 (Stable KER, SKER) A KER E defined on structural types is stable iff, for any two structural types $\mathcal{S}_1$ and $\mathcal{S}_2$:

$$\begin{aligned}
&E(\mathcal{S}_1, \mathcal{S}_2) \ \Rightarrow \\
&\quad E(\mathcal{S}_1, \mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E)) \wedge E(\mathcal{S}_2, \mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E))
\end{aligned}$$

We are now going to instantiate the parametric reduction operator with some different KERs, all of them stable.

5.4 Lossless $\doteq$-driven reduction

Syntactic congruence $\mathcal{T}_1 \doteq \mathcal{T}_2$ (Definition 2) is, essentially, syntactic equality modulo some semantically-irrelevant details. This is a very fine equivalence, hence few types are fused, which results in an inferred type that is potentially very big, but is very precise.

In general, $\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)$ yields a supertype of $\mathcal{T}_1 + \mathcal{T}_2$, which may result in a loss of precision. In this case, since we only fuse structural types that are syntactically congruent, we have a much stronger invariant:

$$\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, \doteq) \simeq \mathcal{T}_1 + \mathcal{T}_2$$

Since $\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, \doteq)$ is not just a supertype of $\mathcal{T}_1 + \mathcal{T}_2$ but is equivalent to the union, the $\doteq$-driven reduction does not entail any loss of type information.

Property 2 (Stability) For any $\doteq$-reduced types $\mathcal{T}_1$ and $\mathcal{T}_2$ and any two $\doteq$-reduced structural types $\mathcal{S}_1$ and $\mathcal{S}_2$, the following properties hold:

$$\mathcal{T}_1 \doteq \mathcal{T}_2 \Rightarrow \mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, \doteq) \doteq \mathcal{T}_1 \doteq \mathcal{T}_2 \quad (1)$$

$$\mathcal{S}_1 \doteq \mathcal{S}_2 \Rightarrow \mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, \doteq) \doteq \mathcal{S}_1 \doteq \mathcal{S}_2 \quad (2)$$

Proof By mutual induction and by cases on the common kind of $\mathcal{S}_1$ and $\mathcal{S}_2$. Property (1): here we observe that every addend of $\circ\mathcal{T}_1$ has one $\doteq$-equivalent addend in $\circ\mathcal{T}_2$, by definition of $\doteq$, and only one, because the two types are $\doteq$-reduced. Hence, the result has one structural addend for each structural addend of $\circ\mathcal{T}_1$, and the two addends are $\doteq$-equivalent by induction. The other

interesting case is the record type case of property (2). Here, by definition of $\doteq$, two record types are only fused when they have exactly the same keys and, for any key k in $Keys(\mathcal{R}_1)$, the types associated to k in $\mathcal{R}_1$ and $\mathcal{R}_2$ are $\doteq$ equivalent, hence, by (1), the type associated in the fused type is equivalent as well. The case for array types is immediate by (1), and the cases for the base types are immediate.

Corollary 1 (Lossless reduction) *For any $\doteq$-reduced types $\mathcal{T}_1$ and $\mathcal{T}_2$:*

$$Reduce(\mathcal{T}_1, \mathcal{T}_2, \doteq) \simeq \mathcal{T}_1 + \mathcal{T}_2$$

Proof The reduction process substitutes, inside $\mathcal{T}_1 + \mathcal{T}_2$, two equivalent addends $\mathcal{S}_1 \doteq \mathcal{S}_2$ with $Fuse(\mathcal{S}_1, \mathcal{S}_2, \doteq)$ which is, by Property 2, syntactically congruent to each of them, hence is $\simeq$-equivalent to each of them, hence is $\simeq$-equivalent to their union.

Example 2 Consider the syntactic-congruence-driven reduction of the following two types: $\mathcal{T}_1 = [\,\{l : \texttt{Num} + \texttt{Str}, m : \texttt{Num}\}\,]$ and $\mathcal{T}_2 = [\,\{l : \texttt{Str} + \texttt{Num}, m : \texttt{Num}\}\,]$. The two types are syntactically congruent, and hence all the corresponding components are syntactically congruent as well. Hence, the two types are explored in parallel and, at each step, the components that correspond are fused, so that the computation of $Reduce(\mathcal{T}_1, \mathcal{T}_2, \doteq)$ proceeds as follows.

$$\begin{aligned}
&Reduce(\mathcal{T}_1, \mathcal{T}_2, \doteq)\\
&\doteq\ Fuse(\mathcal{T}_1, \mathcal{T}_2, \doteq)\\
&\doteq\ [\ Reduce(\{l : \texttt{Num} + \texttt{Str}, m : \texttt{Num}\},\\
&\qquad\qquad \{l : \texttt{Str} + \texttt{Num}, m : \texttt{Num}\}, \doteq)\,]\\
&\doteq\ [\,Fuse(\{l : \texttt{Num} + \texttt{Str}, m : \texttt{Num}\},\\
&\qquad\qquad \{l : \texttt{Str} + \texttt{Num}, m : \texttt{Num}\}, \doteq)\,]\\
&\doteq\ [\,\{l : Reduce(\texttt{Num} + \texttt{Str}, \texttt{Str} + \texttt{Num}, \doteq),\\
&\qquad m : Reduce(\texttt{Num}, \texttt{Num}, \doteq)\}\,]\\
&\doteq\ [\,\{l : Fuse(\texttt{Num}, \texttt{Num}, \doteq) + Fuse(\texttt{Str}, \texttt{Str}, \doteq),\\
&\qquad m : Fuse(\texttt{Num}, \texttt{Num}, \doteq)\}\,]\\
&\doteq\ [\,\{l : \texttt{Num} + \texttt{Str}, m : \texttt{Num}\}\,]
\end{aligned}$$

Consider now the following two types: $\mathcal{T}_1 = [\,\{l : \texttt{Num} + \texttt{Str}, m : \texttt{Num}\}\,]$ and $\mathcal{T}_2 = [\,\{l : \texttt{Str}, m : \texttt{Num}\}\,]$. They share the external structure, but they differ in the type of l, hence they are not $\doteq$-equivalent. Hence, 3.17 the computation of $Reduce(\mathcal{T}_1, \mathcal{T}_2, \doteq)$ does not perform any kind of fusion, but just proceeds as follows:

$$Reduce(\mathcal{T}_1, \mathcal{T}_2, \doteq) \doteq \mathcal{T}_1 + \mathcal{T}_2$$

Example 3 Consider a collection of records whose type is either $\{a : \texttt{Num}, b : \texttt{Num}\}$, $\{a : \texttt{Num}, b : \texttt{Str}\}$ or $\{a : \texttt{Num}, c : \texttt{Str}\}$. The type of the collection is just $\{a : \texttt{Num}, b : \texttt{Num}\} + \{a : \texttt{Num}, b : \texttt{Str}\} + \{a : \texttt{Num}, c : \texttt{Str}\}$.

Example 4 Assume a collection of numeric records r_j, for $j \in J$, each characterized by a set of keys $K_j = \{\!|\, a_i^j \,|\!\}$. Assume that $\mathcal{S}$ collects the different sets of keys that actually appear in the records of the collection, that is, $\mathcal{S} = \{\!|\, K_j \mid j \in J \,|\!\}$. Then, using $\doteq$-reduction, the final type of the collection is

$$\oplus_{K \in \mathcal{S}} \{\, a_i : \texttt{Num} \mid a_i \in K \,\}$$

If S contains very few different key-sets, then this type is small and readable. If the collection is extremely 3.19 heterogeneous, so that, for any two records, they always 1.18 differ in the presence of at least one field, then the type is as big as the collection.

To sum up, when collections are very regular, $\doteq$-reduction yields types that are small and informative. As soon as data becomes less regular, however, this approach, that keeps all non equivalent types separated, produces types that are essentially unreadable. In this case, kind-driven reduction, that we are going to present next, may be better suited.

5.5 Kind-driven reduction

Kind-driven reduction is a reduction driven by the $\mathcal{K}$-equivalence that we defined above. When data are quite irregular, $\doteq$-reduction yields types that are very big, while the adoption of the coarser $\mathcal{K}$-equivalence yields a reasonable result, informative but compact, as shown in Section 7. Stability of $\mathcal{K}$-equivalence is immediate, since the fusion of two types with the same kind yields a type of the same kind by construction.

By definition of $\mathcal{K}$-equivalence, $Reduce(\mathcal{T}_1, \mathcal{T}_2, \mathcal{K})$ fuses every two record types, hence a collection of records always gets a single record type, where the keys that do not appear in every record are marked as optional, and the keys that appear with different types have just one type, obtained by the kind-driven reduction of these different types.

Example 5 Consider again a collection of records whose type is either $\{a : \texttt{Num}, b : \texttt{Num}\}$, $\{a : \texttt{Num}, b : \texttt{Str}\}$ or $\{a : \texttt{Num}, c : \texttt{Str}\}$ as in Example 3. With kind-driven 3.1 reduction, the inferred type for the collection is $\{a : \texttt{Num}, b : (\texttt{Num} + \texttt{Str})?, c : \texttt{Str}?\}$. The final type is smaller than $\{a : \texttt{Num}, b : \texttt{Num}\} + \{a : \texttt{Num}, b : \texttt{Str}\} + \{a : \texttt{Num}, c : \texttt{Str}\}$ but is less informative, since it does not capture the fact that one, and only one, of b and c is always present.

Example 6 Consider again, as in Example 4, a collection of numeric records r_j, for $j \in J$, each characterized by a set of keys $K_j = \{\!|\, a_i^j \,|\!\}$.

Using kind-driven reduction, the final type of the collection is a single record type:

$$\{\ a_i : \texttt{Num}\,\mathtt{q}_i\ \mid\ a_i \in \bigcup_{j\in J} K_j\ \}$$

where $\mathtt{q}_i = !$ iff $i \in \bigcap_{j\in J} K_j$.

As we have already remarked, the type

$$\{\ a_i : \texttt{Num}\,\mathtt{q}_i\ \mid\ a_i \in \bigcup_{j\in J} K_j\ \}$$

owes its compactness to the fact that it treats all fields as independent. In some datasets, the data analyst is actually interested in the co-occurrence of the different record fields, but the syntactic congruence-reduced type is too big. In this case, the following label-driven equivalence will be useful. It is the last one we are going to present.

5.6 Label-driven reduction

When syntactic congruence is too fine but we still want to preserve information about the different field combinations that are present in a collection of records, the following equivalence may be used, that refines $\mathcal{K}$ by specifying that key-sets matter.

1.25 **Definition 10** ($\mathcal{L}(\mathcal{S}_1, \mathcal{S}_2)$) Label equivalence of two structural types $\mathcal{S}_1$ and $\mathcal{S}_2$, written $\mathcal{L}(\mathcal{S}_1, \mathcal{S}_2)$, is defined as follows:

$$\begin{aligned}&\mathcal{L}(\mathcal{S}_1, \mathcal{S}_2)\ \Leftrightarrow\\ &\quad kind(\mathcal{S}_1) = kind(\mathcal{S}_2)\\ &\quad \wedge\ (\ kind(\mathcal{S}_1) = \{\,\}\ \Rightarrow\ Keys(\mathcal{S}_1) = Keys(\mathcal{S}_2)\ )\end{aligned}$$

Label-driven reduction ($\mathcal{L}$-reduction) is not as 'aggressive' as kind-driven reduction ($\mathcal{K}$-reduction): two record types are only merged if they have the same key-sets, but it is not as 'fine' as syntactic congruence-driven reduction: once the key-sets are the same, the record types are merged even if some keys have different types. In this case, the different types associated to a key are recursively reduced. With respect to array types, label-driven reduction always fuses them. More precisely, it always fuses the outermost constructor, and their contents are then actually fused iff they are label-equivalent. In this sense, arrays are treated in the same
1.25 'coarse' way as with kind equivalence. Stability of this equivalence is immediate, since the fusion of two record types with the same keys yields a record type with the same keys.

Example 7 Consider again a collection of records whose type is either $\{a : \texttt{Num}, b : \texttt{Num}\}$, $\{a : \texttt{Num}, b : \texttt{Str}\}$ or
3.1 $\{a : \texttt{Num}, c : \texttt{Str}\}$ (Examples 3 and 5). With label-driven reduction the final type is $\{a : \texttt{Num}, b : (\texttt{Num}+\texttt{Str})\}+\{a : \texttt{Num}, c : \texttt{Str}\}$.

Example 8 Consider again the two types of the second part of Example 2: $\mathcal{T}_1 = [\,\{l : \texttt{Num} + \texttt{Str}, m : \texttt{Num}\}\,]$ and $\mathcal{T}_2 = [\,\{l : \texttt{Str}, m : \texttt{Num}\}\,]$. While $\doteq$-reduction keeps them separated, $\mathcal{L}$-reduction fuses the two types, and we have the following result:

$$Reduce(\mathcal{T}_1, \mathcal{T}_2, \mathcal{L})\ \doteq\ \mathcal{T}_1$$

The type $\mathcal{T}_1 + \mathcal{T}_2$ is actually equivalent to $\mathcal{T}_1$, in the sense that they have the same semantics, hence, *in this specific case*, the size reduction does not entail any information loss.

$\mathcal{L}$-reduction and $\doteq$-reduction may look very similar, since the types of the elements in a collection tend to differ mostly in the presence or absence of some record fields: if we consider Example 4, $\mathcal{L}$-reduction yields in that case exactly the same type as $\doteq$-reduction. Actually, the two relations behave, in practice, very differently, due to the presence of nested records. The following example illustrates this fact through a realistic situation.

Example 9 Consider a massive collection of records r_j with $j \in J$, each with the same set of top-level keys $\{\!|\, a_1, \ldots, a_{10} \,|\!\}$, each r_j containing ten nested records $r^j_1, \ldots, r^j_{10}$:

$$r^j = \{a_1 : r^j_1, \ldots, a_{10} : r^j_{10}\}$$

Assume that, for each i in 1..10, the records in the collection $C_i = \langle\!\langle r^j_i \mid j \in J \rangle\!\rangle$ choose their keys out of a set of ten, eight mandatory and two optional, and assume that each of these collections C_i exhibits all the four possible combinations that derive from the two optional fields. Using label-driven reduction, the resulting type is

$$\{a_1 : \mathcal{T}_1, \ldots, a_{10} : \mathcal{T}_{10}\}$$

where each $\mathcal{T}_i$ is the union of four record types, having $10, 9, 9, 8$ fields. Hence we can say that each $\mathcal{T}_i$ has a total of 36 fields, hence the final type has a total of 10 top-level fields plus $36 * 10$ fields in the nested types: 370 fields. This is four times bigger than the 110 fields (10+10*10) type that would be inferred by $\mathcal{K}$-reduction. If we consider $\doteq$-equivalence, then we observe that this collection may exhibit 4^{10} different types, since each of the 10 top level fields has 4 different possibilities for the associated nested type. Each of these different types contains $10+10*n$ fields, with $8 \leq n \leq 10$, hence,

the resulting type has around $(4^{10}) * (10 + 10 * 9)$ fields, which is more than 10^8. If the fields are independent and the collection is really massive, a big fraction of the 4^{10} possibilities may actually be present in the data. We can make the example more homogeneous by assuming that we have two optional fields at the top level as well. In this case, the sizes of both the $\mathcal{L}$-reduced type and of the $\doteq$-reduced increase by four, hence we have a $\mathcal{K}$-reduced type of size 110, an $\mathcal{L}$-reduced type of size almost 1500 and a $\doteq$-reduced type of size $4 * 10^8$. Hence, in presence of nested records, the $\mathcal{L}$-reduced type can be much more compact than the $\doteq$-reduced type.

5.7 Discussion

We do not believe there is a way to objectively assess the 'best' level of reduction in our applicative scenario, where the type is used as a human-readable formal description of a dataset, since different phases of data exploration need different levels of abstraction. We have applied our techniques to some datasets and our subjective experience is that kind-driven reduction and label-driven reduction are the two forms that are more useful, while the greater detail provided by syntactic congruence seems a bit too fine on realistic datasets. Kind-driven reduction seems best suited for a first look at the data, especially when the dataset is big and not very regular. Label-driven reduction is interesting in order to drill down on label correlation, in a second phase, although it runs the risk of producing types that are too big to be human-readable.

In the experimental section we are going to be more precise about this, producing a synthetic and numerical analysis of our experience.

Many other equivalence relations are interesting, but we believe that the analyst should not be presented with too many options, since this risks to be more confusing than helpful. We believe that the right direction to further increase the flexibility is the definition of an interactive workbench. Examples of different equivalences and a short discussion of the interactive approach are presented in Section 8. 1.1

5.8 Properties

Our reduction operator enjoys three main properties: inclusion, associativity, and commutativity. We formalize them here, while their proofs are in the Appendix. 4.4

Inclusion (Theorem 2) is the fundamental property that specifies that the reduced type includes its two arguments, and ensures, by Theorem 5, that the inferred type is sound. Commutativity and associativity enable an efficient distributed map-reduce implementation, hence ensuring massive scalability.

We start with a technical lemma.

Lemma 1 (Invariant) *The following properties hold, for any SKER E.* 3.8

1. *For any two E-reduced types $\mathcal{T}_1$, $\mathcal{T}_2$,*
   $$Reduce(\mathcal{T}_1, \mathcal{T}_2, E) \text{ is } E\text{-reduced}$$
2. *For any two E-reduced structural types $\mathcal{S}_1$, $\mathcal{S}_2$,*
   $$Fuse(\mathcal{S}_1, \mathcal{S}_2, E) \text{ is } E\text{-reduced}$$

The essential property of the *Reduce* operator is the fact that it returns a supertype of its arguments.

Theorem 2 *For any SKER E and for any two E-reduced types $\mathcal{T}_1$ and $\mathcal{T}_2$:* 3.8

$$\mathcal{T}_1 + \mathcal{T}_2 \leq Reduce(\mathcal{T}_1, \mathcal{T}_2, E)$$

For any two E-reduced structural types $\mathcal{S}_1$ and $\mathcal{S}_2$:

$$E(\mathcal{S}_1, \mathcal{S}_2) \;\Rightarrow\; \mathcal{S}_1 + \mathcal{S}_2 \leq Fuse(\mathcal{S}_1, \mathcal{S}_2, E)$$

The two last fundamental properties of *Reduce* are commutativity and associativity, that enable an efficient distributed map-reduce implementation.

Theorem 3 (Commutativity) *The following two properties hold, for any KER E.*

1. *Given two E-reduced types $\mathcal{T}_1$, $\mathcal{T}_2$, we have:*
   $$Reduce(\mathcal{T}_1, \mathcal{T}_2, E) \doteq Reduce(\mathcal{T}_2, \mathcal{T}_1, E)$$
2. *Given two structural E-reduced types $\mathcal{S}_1$ and $\mathcal{S}_2$ we have:*
   $$E(\mathcal{S}_1, \mathcal{S}_2) \;\Rightarrow\; Fuse(\mathcal{S}_1, \mathcal{S}_2, E) \doteq Fuse(\mathcal{S}_2, \mathcal{S}_1, E)$$

Theorem 4 (Associativity) *The following two properties hold, for any stable KER E.* 3.8

1. *Given three E-reduced types $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$, we have*
   $$Reduce(Reduce(\mathcal{T}_1, \mathcal{T}_2, E), \mathcal{T}_3, E) \doteq Reduce(\mathcal{T}_1, Reduce(\mathcal{T}_2, \mathcal{T}_3, E), E)$$
2. *Given three E-reduced structural types $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ that are mutually E-equivalent, we have*
   $$Fuse(Fuse(\mathcal{S}_1, \mathcal{S}_2, E), \mathcal{S}_3, E) \doteq Fuse(\mathcal{S}_1, Fuse(\mathcal{S}_2, \mathcal{S}_3, E), E)$$

6 Schema Inference

6.1 Schema Inference Rules

We can finally define our type-inference process.

In this paper we use *schema* and *type* as synonyms. Schema inference is the process of inferring a schema (or *type*) for a JSON collection, and is guided by the rules described in Figure 6. The judgments are parametrized over a KER E, which is used by the *Reduce* operator. Each rule allows one to infer the type of the value indicated in the conclusion (part below the line) in terms of types recursively determined in the premises (part above the line). Rules with no premises specify the terminal case of the process. Judgment $\vdash^E J : \mathcal{S}$ states that a JSON expression J has a structural type $\mathcal{S}$, while $\vdash^E J_1, \ldots, J_n :^c \mathcal{T}$ states that $\mathcal{T}$ is a type for the collection $J_1, \ldots, J_n$.

The type rules transform atomic values and records into the corresponding types in the obvious way. Elements of an array are treated as a collection, and the collection type becomes the internal type of the array.

The rule (TypeCollection) is the core of our approach. It requires the input collection to be split until singletons are met. At that point rule (TypeSingle) invokes elementary type inference, and then the results are combined using the merging function *Reduce*. The rule is non-deterministic, but the final result does not depend on the choice of the split point since *Reduce* is associative.

The following lemma states that every type that is inferred is always E-reduced, which specifies a formal relation between type inference and the E equivalence upon which it depends.

Lemma 2 (Invariant) *For any JSON expressions J, J_i, for any structural type $\mathcal{S}$, type $\mathcal{T}$, SKER E, the following properties hold.*

1. $\vdash^E J : \mathcal{S} \Rightarrow \mathcal{S}$ *is E-reduced*
2. $\vdash^E J_1, \ldots, J_n :^c \mathcal{T} \Rightarrow \mathcal{T}$ *is E-reduced*

The following theorem states soundness of typing. We would like to stress that, although we focus on a specific *Reduce* function, soundness holds for any choice of $Reduce(\mathcal{T}_1, \mathcal{T}_2, E)$ that satisfies the inclusion property of Theorem 2, while determinism of the result only depends on associativity.

Theorem 5 *For any SKER E, for any JSON expressions $J, J_1, \ldots, J_n$:*

$$\vdash^E J : \mathcal{S} \Rightarrow [\![J]\!] \in [\![\mathcal{S}]\!]$$
$$\vdash^E J_1, \ldots, J_n :^c \mathcal{T} \Rightarrow \{\!\{ [\![J_1]\!], \ldots, [\![J_n]\!] \}\!\} \subseteq [\![\mathcal{T}]\!]$$

Proof We prove it by mutual induction on the size of the inference proof and by cases on the last applied rule. The base rules are trivial. The cases for the record and array rules are an immediate consequence of the semantics of records and arrays. The empty collection rule is trivial and the singleton rule follows immediately by induction. For the crucial (TypeCollection) rule, we know by induction that

$$\{\!\{ [\![J_1]\!], \ldots, [\![J_i]\!] \}\!\} \subseteq [\![\mathcal{T}_1]\!]$$
$$\{\!\{ [\![J_{i+1}]\!], \ldots, [\![J_n]\!] \}\!\} \subseteq [\![\mathcal{T}_2]\!]$$

By Theorem 2,

$$\mathcal{T}_1 \leq Reduce(\mathcal{T}_1, \mathcal{T}_2, E) \text{ and } \mathcal{T}_2 \leq Reduce(\mathcal{T}_1, \mathcal{T}_2, E)$$

Hence, by transitivity, we have that

$$\{\!\{ [\![J_1]\!], \ldots, [\![J_i]\!] \}\!\} \subseteq [\![Reduce(\mathcal{T}_1, \mathcal{T}_2, E)]\!]$$
$$\{\!\{ [\![J_{i+1}]\!], \ldots, [\![J_n]\!] \}\!\} \subseteq [\![Reduce(\mathcal{T}_1, \mathcal{T}_2, E)]\!]$$

hence

$$\{\!\{ [\![J_1]\!], \ldots, [\![J_n]\!] \}\!\} \subseteq [\![Reduce(\mathcal{T}_1, \mathcal{T}_2, E)]\!].$$

It is worth noticing that schema inference does not directly exploit the full expressiveness of the schema language: for example, optional types and union types do not appear in the inference rules. These operators are only introduced by the *Reduce* function.

7 Experimental Evaluation

This section presents the results of the experimental evaluation that we performed to assess our claims on precision, concision, and efficiency.

7.1 Implementation

Our implementation is based on Apache Spark, a well-adopted framework for the parallel processing of large datasets. One of the most appealing features of Spark is its ability to keep large datasets into main memory, which enables the fast processing of complex analytic tasks. We implemented our tool by using the functional programming language Scala, as it makes the definition of inductive operators and the use of higher order functions quite easy. Scala allowed us to implement both the type inference and the type fusion algorithms in a straightforward manner starting from their respective formal specifications.

To run our inference technique on the Spark cluster, we used a program consisting of a *map* transformation that infers a type for each input JSON object; a *reduce* action is then used to merge the resulting types.

$$\frac{}{\vdash^E \textbf{null} : \texttt{Null}}\ (\textsc{TypeNull}) \qquad \frac{}{\vdash^E \textbf{true/false} : \texttt{Bool}}\ (\textsc{TypeBool}) \qquad \frac{n \in \textbf{Number}}{\vdash^E n : \texttt{Num}}\ (\textsc{TypeNumber}) \qquad \frac{s \in \textbf{String}}{\vdash^E s : \texttt{Str}}\ (\textsc{TypeString})$$

$$\frac{\forall i.\ \vdash^E J_i : \mathcal{S}_i \qquad \forall i,j.\ i \neq j \Rightarrow l_i \neq l_j}{\vdash^E \{l_1 : J_1, \ldots, l_n : J_n\} : \{l_1 : \mathcal{S}_1!, \ldots, l_n : \mathcal{S}_n!\}}\ (\textsc{TypeRec}) \qquad \frac{\vdash^E J_1, \ldots, J_n :^c \mathcal{T}}{\vdash^E [J_1, \ldots, J_n] : [\mathcal{T}]}\ (\textsc{TypeArray})$$

$$\frac{}{\vdash^E \emptyset :^c \emptyset}\ (\textsc{TypeEmp}) \qquad \frac{\vdash^E J : \mathcal{S}}{\vdash^E J :^c \mathcal{S}}\ (\textsc{TypeSingle}) \qquad \frac{\vdash^E J_1, \ldots, J_i :^c \mathcal{T}_1 \quad \vdash^E J_{i+1}, \ldots, J_n :^c \mathcal{T}_2 \quad n \geq 2, \quad 1 \leq i \leq n-1}{\vdash^E J_1, \ldots, J_n :^c \mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)}\ (\textsc{TypeCollection})$$

Fig. 6 Type inference rules.

7.2 Runtime environment

We performed our experiments on a (relatively small) cluster, consisting of five slave nodes used for storing and processing the datasets and an additional master node to coordinate them. Each node is equipped with two 2.2 GHz 64-bit Intel Xeon CPUs of 10-cores each, 64GB of main memory, and a 1TB RAID hard drive. We used HDFS 2.7 for the distributed storage of our 1.3 datasets, and Spark 2.3.0 as the computation engine. To take full advantage of the cluster capacity, we used all its cores, that is 5*2*10=100, and set the memory to 60 GB per node in all our experiments.

7.3 Datasets

1.2 To evaluate our approach we used seven real life datasets: 4.8 three datasets that have been already used in [7], a more recent version of the Wikidata than the one initially used in [7], and three new datasets that were obtained from new sources; these datasets are listed below.

- The GitHub dataset [19] containing objects corresponding to *pull requests* metadata generated upon each call to the GitHub web service. This dataset does not use arrays, but it represents a good testbed for measuring the effect of the presence of different sets of fields in different records.
- The Twitter'16 dataset [19] containing *tweet* records describing metadata about tweets, as well as a few *delete* records, which correspond to metadata generated upon delete requests. This dataset is interesting because it mixes two kinds of objects, but also because it uses arrays quite intensively.
- The NYTimes dataset, crawled using the NYTimes *articlesearch* API call, containing metadata about the NYTimes articles.[2] An interesting feature of this dataset is the variability of the structure attached to the same field in different instances, which distinguishes it from the previous datasets.
- The Wikidata dataset, that comprises more than 43 million records (501GB total size), and was obtained from the official repository by taking the 16/06/2018 snapshot.[3] Being a JSON representation of the Wikimedia knowledge base, the objects in this dataset follow a rather regular structure and do not contain many variations of the structure of the same field as in the NYTimes dataset. However, the dataset violates the basic principle of schema design, that is, the distinction between data and structure. For instance, in many fields the keys carry some information about the instances they describe (like identifiers or language code) instead of encoding the information as the value of a specific field. This design choice has a dramatic impact on the size of the inferred schema (whether we use our technique or the Spark one) which becomes very large as it is not possible to rely on field keys to fuse similar types. Hence, this dataset is quite interesting since it allows us to study the limitations of our approach, for example the limits of the size of the schemas that we can manage.
- The Twitter'18 dataset, obtained from the Kaggle repository, and containing *tweet* records collected during the 2018 Russian election campaign.[4] This dataset uses a more recent API version than the Twitter'16 one, and this is reflected by the increase of the average number of fields in the records (39 versus 23 in Twitter'16). This dataset is interesting due to the existence of many optional fields.

[2] `https://developer.nytimes.com`

[3] `https://dumps.wikimedia.org/wikidatawiki/entities/`

[4] `https://www.kaggle.com/borisch/russian-election-2018-twitter`

– The VK dataset, also obtained from Kaggle and related to the 2018 Russian election.[5] It contains records describing interactions of users in the VK social networking site and, like the NYTimes dataset, features some structural variations that are not fully described in the API Documentation.[6]
– The Core dataset, which is the largest one (508 GB), obtained from the Core website, exposing information about research articles aggregated from many open repositories worldwide.[7] The current dataset corresponds to the dump of March 1st, 2018, which describes more than 123 million articles. The records in this dataset follow a fixed schema that is documented in the website[8] quite precisely except in the case of arrays containing records which are approximated to arrays of strings; this shows that even in very regular and well-documented datasets one can still discover that the actual structure of data is different from what is documented. This dataset serves mostly to study a situation with a very large dataset having a very regular schema.

These datasets are further described in Table 1, which reports basic statistics such as their size in GB, the number of their objects, the average size of the textual representation of each of these objects in KB, the average size of the corresponding abstract syntax tree (AST), and the average height of the AST.

7.4 Spark Dataframe Type inference

Spark is endowed with a schema inference mechanism that can be invoked on an object of the Dataframe class. This mechanism is efficient and somehow precise as it allows one to reveal, for each record, the set of its fields with their associated types and, for each array, a compact representation of its type; this compact representation is obtained by a fusion mechanism that bears many similarities with our kind equivalence approach. However, the lack of union types prevents Spark from precisely representing the situation in which a field uses a different structure in different instances, forcing Spark inference to resort to over-approximations. Another major limitation of Spark is the impossibility to distinguish between mandatory and optional fields, despite the presence of a *nullable* flag. We observed that the value of this flag is always set to true, and an inspection of the Spark source code allowed us to confirm that this truth value is never updated. Due to the lack of a clear documentation, we illustrate the main difference between our technique and the Spark inference by using a sample dataset consisting of three records sharing three fields: one of them (the *coord* field) has a different structure in each record, and another one (*email*) is present in one record only.

```
{first: "al", last: "jr", coord:[ ]}

{first:"al", last:"jr", coord: null}

{first: "li", last: null, coord: {long: 12,
    lat: 45}, email: "abc@ef"}
```

Thanks to union types, the inferred $\mathcal{K}$-type captures the structure variability of *coord* as follows:

```
{first:Str, last:Str+Null, coord: [ ]+ Null +
    {long:Num, lat:Num}, email?: Str}
```

Spark, instead, which does not use union types, coerces the type of *coord* to an array of Strings and raises an error by dubbing the inferred record as *corrupted*, as shown in the textual representation below:

```
root
|-- _corrupt_record: string (nullable = true)
|-- coord: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- first: string (nullable = true)
|-- last: string (nullable = true)
```

The *containsNull* flag, which in principle should pinpoint the presence of null values inside arrays, is always set to true, hence it is almost useless.

Hence, the Spark inference engine cannot infer many pieces of information that our technique recovers. To measure the extent of this information loss, we report in Table 2, for each dataset, the number of union types and of optional fields that are inferred by the $\mathcal{K}$-reduction and which are missing in the types inferred by Spark. For all the datasets, except Wikidata and Core, the information loss is quite important.

While this loss of precision is the main difference between Spark approach and our $\mathcal{K}$-reduction, we should not forget that the main difference among our approach and Spark approach is parametricity: we can choose the precision level, while Spark is bound to one choice only.

7.5 Efficiency analysis

To evaluate the efficiency of our approach, we measured the execution time of the inference algorithm for both equivalences, and compared it with that of the Spark Dataframe inference; the findings we collected are reported in Table 1. The measured time is the total time required to: i) load the data from HDFS, ii) run the inference on each data partition in parallel and iii) send

[5] https://www.kaggle.com/borisch/russian-election-2018-vkcom-user-activity/feed
[6] https://vk.com/dev/streaming_api_docs_2
[7] https://core.ac.uk
[8] https://core.ac.uk/services#dump-structure

Dataset	GitHub	Twitter'16	NYTimes	Wikidata	VK	Twitter'18	Core
Datasets description							
Size	13.7 GB	21 GB	21.3 GB	501 GB	5.2 GB	10.7 GB	508.7 GB
# objects	1,000,001	9,901,087	1,184,943	49,013,568	3,036,654	1,945,365	123,986,577
average textual size	14.7 KB	2.2 KB	19.3 KB	11 KB	1.4 KB	5.5 KB	4.4 KB
average AST size	495.46	138.67	1,165.17	878.9	51.38	353.85	54
average AST height	5.0	3.77	5.97	8.4	4.63	5.84	4
Succinctness Results							
$\mathcal{K}$-reduction							
Map: avg. type size	495.46	135.44	109.74	643.6	48.7	337.5	44.4
Red.: final type size	655	559	139	907,876	554	2,194	83
Red./Map size ratio	1.3	4.12	1.3	1,410	11.4	6.5	1.9
$\mathcal{L}$-reduction							
Map: avg. type size	495.46	135.44	128.54	646.7	49.2	338.5	44.4
Red.: final type size	2,979	2,438	384	failed	20,314	130,891	83
Red./Map size ratio	6	18	3	NA	412	387	1.9
Spark Dataframe Inference							
inferred type size	627	513	94	908,471	549	2,142	67
Total execution time							
$\mathcal{K}$-reduction	60 sec	120 sec	111 sec	37.6 min	36 sec	43 sec	26.9 min
Throughput	234 MB/sec	179 MB/sec	197 MB/sec	227 MB/sec	148 MB/sec	255 MB/sec	322 MB/sec
$\mathcal{L}$-reduction	65 sec	132 sec	123 sec	failed	38 sec	50 sec	27.2 min
Spark Dataframe	46 sec	92.5 sec	84 sec	25.8 min	29 sec	27 sec	21.5 min
$\mathcal{K}$-red. vs Spark DF	30 %	29.8 %	32 %	54.8 %	24 %	59 %	25 %

Table 1 Original dataset description, succinctness results, and total execution times.

Data-sets	GitHub	Twitter'16	NYTimes	Wikidata	VK	Twitter'18	Core
Number of union types	28	46	12	1	5	55	17
Number of optional fields	17	29	28	87,673	104	234	0
Total number of fields	325	278	70	528,027	293	1,179	36

Table 2 Quantification of the amount of structural information that is lost going from $\mathcal{K}$-reduction to Spark Dataframe inference.

the inferred types to the *driver* node where they are merged. We managed to process all datasets except the Wikidata for which the $\mathcal{L}$-reduction raised an out-of-memory exception due to the very large number of different key-sets that this dataset contains.

We noticed that the execution time is dominated by the time to load and parse the data (steps i and ii) whereas the time to merge the types (step iii) is very small. This explains why our technique, implemented in Scala with no special optimization, is not very far in speed from the Spark inference, which is native. In detail, the average time increase of $\mathcal{K}$-reduction over Spark inference is 36%, and we infer more precise schemas, as we discussed. The increase of time from $\mathcal{K}$-reduction to $\mathcal{L}$-reduction is also quite limited, typically around 10%.

7.6 Succinctness analysis

Both tested equivalences produce types that are quite succinct, and, of course, $\mathcal{K}$-reduction produces more compact types than $\mathcal{L}$-reduction.[9]

[9] The inferred types for each dataset are reported in [1].

We analyze succinctness by comparing the size of the types that we obtain with the size of the objects. Some form of size reduction is already obtained upon the map phase, where each single object is analyzed, by the fusion of equivalent types that are found inside an array. The most prominent example of this is observed for NYTimes: the AST size of the average type is around 9% of the size of the AST of the average object (using $\mathcal{K}$-reduction). This is due to the presence of arrays, which, in this dataset, typically contain 10 objects, whose type is represented by just one type. On the other extreme, for the GitHub dataset, the average AST size of the types is the same as the average AST size of the records, since here we have no array.

The results after the reduction phase are more interesting. If we ignore the Wikidata dataset, which features a serious design issue, the size of the resulting type for $\mathcal{K}$-reduction is, in the worst case, 11.4 times bigger than the average size of the single types inferred during the map phase (VK dataset). For the other datasets, this increase is even smaller, meaning that JSON value types are very similar, and that the type that describes

the entire collection is manageable, and hence can be reasonably read and understood by the data analyst.

If we consider $\mathcal{L}$-reduction, the size of the resulting types for VK and Twitter'18 which feature the worst case are respectively 412 and 387 times bigger than the average size of the single map phase types. Hence, in this case, the resulting type is not very easy to read, because of its size, but it is very informative, as it contains very precise information about the different field combinations, and can be handled in main-memory. For the other datasets, the increase remains smaller and the resulting type is more manageable.

As already observed, it is in general impossible to decide a priori whether the information gain of the $\mathcal{L}$-reduction is worth the size increase, since the information about field correlations may be of great interest to the user, hence worth the effort, or it may be totally irrelevant, depending on his/her need.

7.7 Precision analysis

Despite the unavoidable subjectivity of the 'precision' question, one can try and quantify the information gained going from the $\mathcal{K}$-reduced type to the $\mathcal{L}$-reduced type, as follows. A record type inferred by $\mathcal{K}$-reduction with 10 different optional fields a,b,...,i,j, may describe two different extreme situations, one where we only have 2 different actual record shapes, that is two different keysets that actually appear in the data (say, $\{\!\{\ a\ \}\!\}$ and $\{\!\{\ b,\ldots,i,j\ \}\!\}$), and another extreme situation where every subset of $\{\!\{\ a,b,\ldots,i,j\ \}\!\}$ describes at least one actual record shape in the data, hence we have 2^{10} different shapes.

In the '2 shapes' situation, the passage from the $\mathcal{K}$-reduced type to the $\mathcal{L}$-reduced type is definitely interesting. In this case, the $\mathcal{L}$-reduced type would be a union type with just two addends, and its total size would be essentially the same as the size of the $\mathcal{K}$-reduced type, but the gained knowledge would be great: in the $\mathcal{L}$-reduced type we see how the fields are partitioned in the two kinds of records. In this case we have a very strong correlation between keys: given two keys, either the presence of one implies the presence of the other, or it excludes it.

In any situation that approximates the '2^{10} shapes' situation, on the contrary, the passage is hardly worth the effort: the $\mathcal{L}$-reduced type is (almost) three orders of magnitudes bigger than the $\mathcal{K}$-reduced type, which makes it unreadable, and, moreover, the presence of a huge number of different sets of labels implies that label correlation is, generally, weak, and is difficult to describe and understand.

This observation is obviously affected by the same subjectivity problems that we previously described, but is indeed related to a practical observation: a $\mathcal{K}$-reduced type with n optional fields indicates the possibility of 2^n shapes, an $\mathcal{L}$-reduced type with a union of k different record types indicates the actual presence of k shapes, hence the distance between 2^n and k is an indication of the information that is gained by going from the first type to the second.[10]

To examine our types with respect to this notion of precision, we compared, for each record type found in each dataset, the total number of fields, the number of *optional* fields computed from the $\mathcal{K}$-reduced type, the *maximal* number of its shapes according to this number, and the *actual* number of such shapes obtained from the $\mathcal{L}$-reduced type. Some representative examples of these numbers are reported in Table 3: every column in the table describes one record type that is found in the $\mathcal{K}$-reduced type of the corresponding dataset, compared with its expanded version in the $\mathcal{L}$-reduced type. For each dataset we have chosen some record types that are representative of typical situations.

To better explain the meaning of these numbers, consider the last column of the NYTimes dataset, which describes the case of a record type containing 9 String fields all of which are optional and whose $\mathcal{K}-$type and $\mathcal{L}-$type are given below while omitting the basic type of each field.

$\mathcal{K}-$type

```
{thumbnailwidth ?, thumbnailheight ?,
    thumbnail ?, widewidth ?, wideheight ?,
    wide ?, xlargewidth ?, xlargeheight ?,
    xlarge ?}
```

$\mathcal{L}-$type

```
{thumbnailwidth, thumbnailheight, thumbnail} +
{widewidth, wideheight, wide } +
{xlargewidth, xlargeheight, xlarge}
```

The $\mathcal{K}$-reduced type is compatible with $2^9 = 512$ combinations of the fields in this record, while the $\mathcal{L}$-

[10] We may be more formal, as follows: consider n keys and a space where every point is a set of shapes, that is, a set of subsets of $\{\!\{\ 1,\ldots,n\ \}\!\}$. In this setting, every $\mathcal{L}$-reduced type exactly indicates one point of a space whose size is 2^{2^n}, hence each $\mathcal{L}$-reduced type brings exactly the same amount of information, 2^n bits. On the other side, a $\mathcal{K}$-reduced type is, in general, compatible with many different points in this space, hence it brings a lower number of bits, that depends on the number of optional keys, and may be computed for each specific $\mathcal{K}$-reduced type. We may compare this number with 2^n in order to mathematically quantify the information gain. We do not pursue this avenue because this model embeds the unrealistic idea that every distribution of shapes has the same probability, and because we do not believe that this model, although mathematically coherent, is a useful model of the information needs of the data analyst.

	Github				Twitter'16				NYTimes						VK						Twitter'18					
number of optional fields in the record	1	2	2	4	2	3	3	13	1	1	3	3	5	9	13	9	5	2	8	3	9	7	3	1	3	3
total number of fields of the record	17	6	30	9	42	3	3	23	21	4	5	9	6	9	14	17	12	3	18	8	39	7	4	3	4	4
maximal number of shapes	2	4	4	16	4	8	8	8192	2	2	8	8	32	512	8192	512	32	4	256	8	512	128	8	2	8	8
exact number of shapes	2	3	3	2	3	4	5	5	2	2	4	5	10	3	13	13	32	4	56	4	40	102	3	2	2	4

Table 3 Comparison of the information obtained from $\mathcal{K}$ reduction and $\mathcal{L}$ reduction. Every column describes one record type that is found in the dataset.

reduced type describes the actual shapes that appear, that are just 3.

This example also shows the practical interest of the extracted schemas. The first schema allows the user to know which fields may be present in the data, which is essential in order to know which kind of information can be extracted and may also become the basis for some well-known types of optimization. The second schema gives a much richer information, showing that the different fields are always grouped in a way that is extremely regular and that may help the analyst understand their meaning. 4.6, 4.7

From Table 3 we notice that, in general, the two extreme cases that we cited before ($n >> 1$ optional fields with either 2^n actual shapes, or 2 actual shapes only) in practice never happen, hence there is no situation where one choice clearly dominates the other one. We observe that the percentage of optional fields has a lot of variability, going from a 5% up to a 100%. We notice that the number of different shapes for each record type is generally low but not negligible: we have a lot of situations with 4 or 5 different shapes, which indicates a size expansion in the type that is not a priori unbearable but that has a cost that must be justified by the needs of the data analyst. On the other side, there are some situations with a high number of optional fields and a low number of actual shapes, where the expansion from the $\mathcal{K}$-reduced type to the $\mathcal{L}$-reduced type may be quite informative, such as the NYTimes example that we discussed before. Such situations occur in almost every dataset.

We are aware of the existence of mathematical approaches to establish an 'optimum' balance between size and precision for an inferred schema, such as the one based on the *Minimum Description Length* used in [21] for the DTD inference of an XML document, but our inspection of these real-world datasets seems to indicate that such a mathematical 'optimum' does not make sense here: the optimal choice is typically not objective but depends on the current needs of the analyst. We believe that, in a typical situation, a data analyst may use the $\mathcal{K}$-reduced type in order to gain a first understanding of data structure, and may then use the $\mathcal{L}$-reduced type to get a better understanding on some specific record types.

7.8 Scalability analysis

To assess the scalability we need to use datasets of increasing size. While the easiest way to build such datasets is by concatenating the original ones several times, this *multiplication* process increases the homogeneity of the datasets, which, in principle, may influence the result; this suggests the adoption of a *division* approach, where the testbed comprises bigger and bigger fractions of the original dataset, such as the $20\%, 40\%, 60\%, 80\%, 100\%$. As we wanted to extend the scale of our experiments up from the size of our datasets, we favour a *division+multiplication* approach, where we used both the five fractions above and the datasets obtained by copying the original datasets from 2 up to 10 times.[11] We do not include the Wikidata and the Core datasets in this analysis as they are already large enough and suffice to show that our approach scales to 500 GB, at least with the $\mathcal{K}$-reduction.

The execution times for the remaining datasets are reported on Figure 7. We first observe that the graphs 1.23
are sub-linear in the division area and linear in the multiplication area, showing that the algorithm scales with the size of the analyzed data. The sub-linearity in the division area can be explained by the architectural design of Spark which is impacted by the data organization in the HDFS storage: when the dataset to process is small enough to fit in fewer blocks than the number of available cores, Spark uses only one *wave* of execution to process this dataset, which avoids any overhead due to the synchronization between subsequent executions waves. All the datasets we obtained using the division approach have been processed within one wave of execution, which explains the sub-linearity, as shown in

[11] In the case of VK we multiplied the original datasets 4, 6, ..., 20 times to reach a minimum size of 100 GB as the largest size.

Figure 7 (f), where we focused on the "division" zone for the VK dataset.

We also notice that the overhead of our algorithm with respect to the native Spark algorithm grows with the size increase no matter the dataset at hand. This growth is entirely due to the data loading phase which, again, is better dealt with in the Spark native inference.

One final observation that we draw from an inspection of the execution log is that our algorithm uses the full computational resources of the cluster (the 100 cores of all node) when processing our datasets, which explains its excellent performance.

8 Extensions

In this work we have presented a general framework for the distributed inference of type information about JSON data. We kept the base type system as simple as possible, but there are many extensions that would be quite interesting in practice, such as:

- quantitative types;
- constraints over simple types;
- enumeration types;
- variant types;
- tuple types;
- keyset equivalence;
- interactive summarization;
- difference types.

We give here a brief introduction for each of them.

Quantitative types A distributed process of type inference may collect quantitative information about each variant of the data and about the average, minimum, or maximum length of the arrays. This is the most interesting extension of the mechanism that we present here, and is presented in detail in [9].

Constraints over Simple Types JSON schema [5] allows one to define some constraints on the values of simple types, such as size bounds for numbers, being a multiple of a specific integer for integers (it distinguishes between *integers* and *numbers*), length bounds for strings, adherence to a specific format or to a regular expression for strings. Each of this constraints is, in practice, quite useful for somebody who needs to interpret a dataset, and most of them are quite trivial to integrate in our framework, with the notable exception of regular expressions.

For example, one may integrate most constraints by, first of all, extending the syntax of base types with constraints indicators, as follows.

$$
\begin{array}{rll}
\mathcal{B} & ::= \texttt{Null} \mid \texttt{Bool} \mid \mathcal{BN} \mid \mathcal{BS} & \textbf{Basic types} \\
\mathcal{BN} & ::= \texttt{Num}(i{:}j) \mid \texttt{Num}(i{:}j, \mathit{Factors}(S)) & \textbf{Numeric types} \\
\mathcal{BS} & ::= \texttt{Str}(i{:}j, \mathit{Formats}(S)) & \textbf{String types}
\end{array}
$$

Individual type inference would first associate a number n to the trivial bounds $(n : n)$, and an integer i to the set of its factors: $i : \texttt{Num}(i{:}i, \mathit{Factors}(\mathit{factorize}(i)))$.

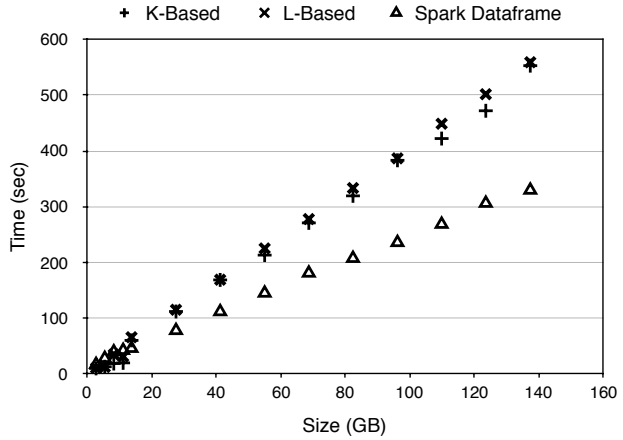
The fusion function, whenever two types of the *numeric* kind are fused, would compute the minimal upper bound, with respect to the subtype relation, that is, the strongest constraint that is implied by both. Hence, the upper bound between `Num` and `Num` is `Num`. The upper bound between two size intervals is the smallest interval that includes both of them. Finally, the factors multisets would be intersected when two types are merged, in order to only keep the common factors. This process is commutative, associative, and sound (meaning that it yields a supertype of the merged types), and very efficient to implement.

Lengths of strings can be computed in the same way and, if a fixed set of formats is provided, it is also simple to compute in a distributed way the subset of formats that a set of strings satisfies. The problem of the distributed inference of a useful regular expression is much more difficult, and we leave it open for future research.

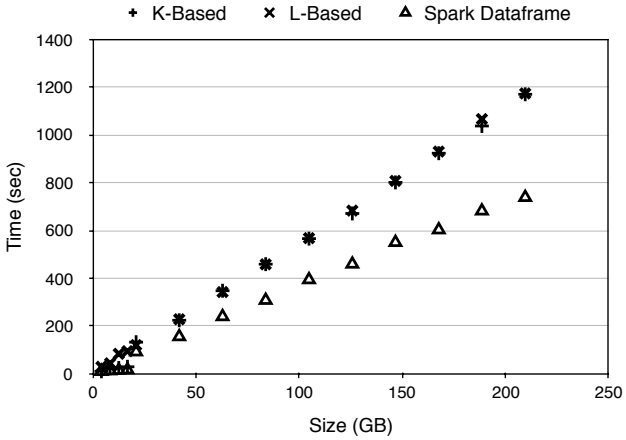
Enumeration Types In our exploration, we found that it is quite common to have fields that only assume few different values, and this is a very valuable piece of information. The formal description of an enumeration type with atomic values is very simple – we would just extend the corresponding kind with an enumeration type.

$$
\begin{array}{rll}
\mathcal{B} & ::= \texttt{Null} \mid \texttt{Bool} \mid \mathcal{BN} \mid \mathcal{BS} & \textbf{Basic types} \\
\mathcal{BN} & ::= \texttt{Num} \mid \mathit{NumEnum}\{n_1, \ldots, n_j\} & \textbf{Numeric types} \\
\mathcal{BS} & ::= \texttt{Str} \mid \mathit{StrEnum}\{s_1, \ldots, s_j\} & \textbf{String types}
\end{array}
$$

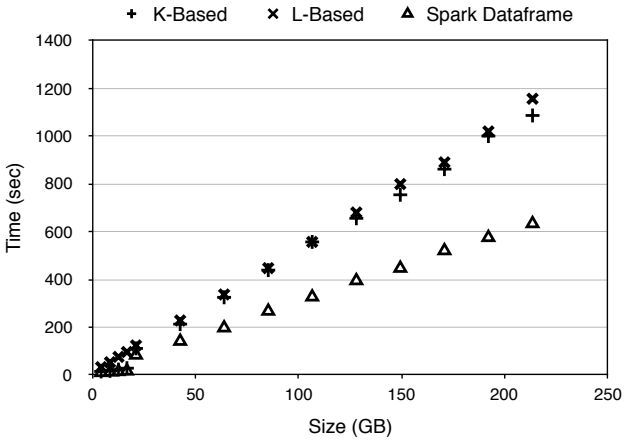
Consider for example string types. We would consider all string types as mutually equivalent, hence they would always be fused. For any string s we would first infer the type $\mathit{StrEnum}\{s\}$. Fusion would depend on a threshold k: the fusion of two enumeration types would produce their union if the size of the result is less than k, while it would produce the generic type `Str`, possibly enriched with constraints, when the size is above k. The fusion of `Str` and an enumeration type would yield `Str`. Once again, the process is clearly commutative, associative, and sound.



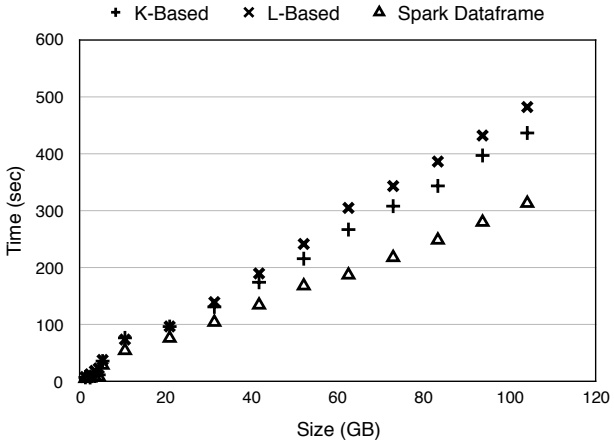
(a) Github



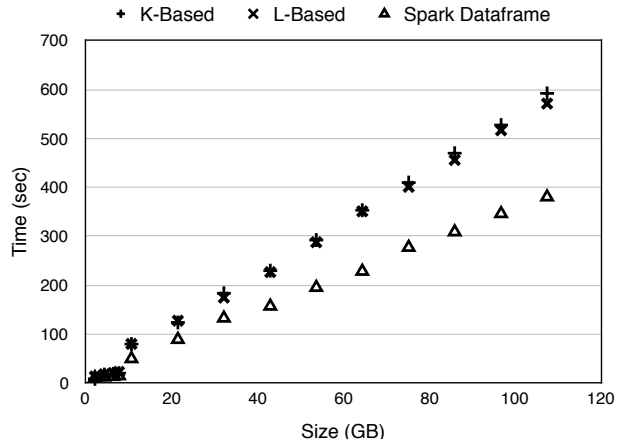
(b) Twitter



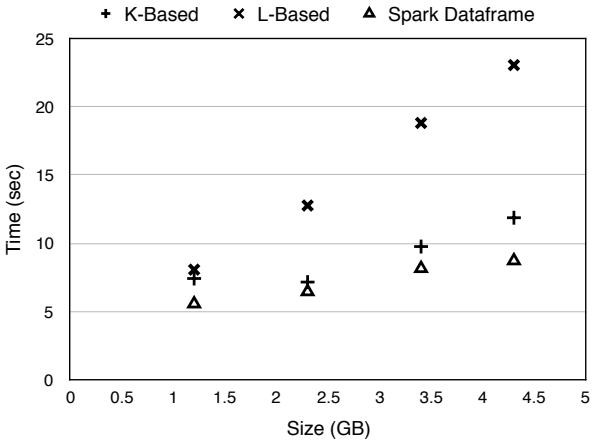
(c) NYTimes



(d) VK 2018



(e) Twitter 2018



(f) VK 2018 close-up for the "division" experiment

Fig. 7 Scalability results.

Variant Types Variant types are a common jargon where, in a record, the value of a discriminator field dictates the structure of the rest of the record. We can easily express this fact using enumerated types and union types, as in the following example.

$$\{Type: StrEnum\{''person''\}, Name: \texttt{Str}, Age: \texttt{Num}\} + \{Type: StrEnum\{''org''\}, Name: \texttt{Str}, Address: \texttt{Str}\}$$

Our algorithm, extended with enumeration types, would indeed infer that type if we used $\mathcal{L}$-equivalence or $\doteq$-equivalence. If we use $\mathcal{K}$-equivalence, hence declaring that we are not interested in distinguishing different label combinations, our algorithm would infer the following type instead.

$$\{Type: StrEnum\{''person'',''org''\}, Name: \texttt{Str}, Age: \texttt{Num}?, Address: \texttt{Str}?\}$$

Hence, enumeration types seem to provide an interesting approach to the problem of variant types.

Tuple Types In type theory, *tuple* denotes an ordered structure where the meaning of the components, hence their type, is not indicated by a label but by their position.

In JSON, there are data sets where records such as $\{Name: ''John'', Age: 23\}$ are systematically encoded as arrays $[''John'', 23]$: such an array is used as a *tuple*: the size is fixed, and the position encodes the meaning of the field. This is for example the case for a list of coordinates stored as $[[1,1],[1,2],[2,3]]$, rather than $[\{x:1,y:1\},\{x:1,y:2\},\{x:2,y:3\}]$.

In order to capture tuple types, we could consider, as in JSON schema, two different classes of array types, as in the following syntax.

$$\mathcal{A} ::= [\mathcal{T}*] \mid [\mathcal{T}_1, \ldots, \mathcal{T}_n] \quad \textbf{Array Types}$$

Here, a type $[\mathcal{T}*]$ is our current array type, that allows arbitrary repetitions of $\mathcal{T}$, while $[\mathcal{T}_1, \ldots, \mathcal{T}_n]$ only allows exactly those types in exactly that order, hence being a subtype of the weaker type $[(\mathcal{T}_1 + \ldots + \mathcal{T}_n)*]$, that imposes no order, and allows each $\mathcal{T}_i$ to appear an arbitrary number of types. The distributed inference is not very difficult, but we will not illustrate it here for reasons of space.

Keyset Equivalence The $\mathcal{L}$-equivalence is often too verbose while the $\mathcal{K}$-equivalence hides any correlation information. It is often the case that one is interested in the correlation of two or three keys only, and would not like to go to the full size of $\mathcal{L}$-equivalence. In this case, we may define a notion of Keyset-equivalence, parametrized over a set of keys K, such that two record types are identified iff they coincide with respect to the presence of the keys in K: 1.1, 3.26

Definition 11 ($\mathcal{KS}_K(\mathcal{S}_1, \mathcal{S}_2)$) Keyset-equivalence of two structural types $\mathcal{S}_1$ and $\mathcal{S}_2$, with respect to a set of keys K, written $\mathcal{KS}_K(\mathcal{S}_1, \mathcal{S}_2)$, is defined as follows:

$$\begin{aligned}&\mathcal{KS}_K(\mathcal{S}_1, \mathcal{S}_2) \Leftrightarrow \\ &\quad kind(\mathcal{S}_1) = kind(\mathcal{S}_2) \\ &\quad \wedge\ (\ kind(\mathcal{S}_1) = \{\,\} \Rightarrow \\ &\qquad\qquad (Keys(\mathcal{S}_1) \cap K) = (Keys(\mathcal{S}_2) \cap K)\ )\end{aligned}$$

Hence, the analysis of a collection of records with respect to $\mathcal{KS}_{\{\!|\ a,b\ |\!\}}$ would show the correlation between a and b: it would create four different groups if all the combinations of (a,b) are present, just two groups $\{a: \ldots, \ldots\} + \{b: \ldots, \ldots\}$ if they are alternative, and so on. 1.1

The amount of intermediate relations between $\mathcal{K}$ and $\mathcal{L}$ is infinite. One may also consider, for example, a refinement of $\mathcal{KS}_K$ where all keys of K must also be associated to types with the same set of kinds, and this may be iterated for a fixed number of levels. 1.1

While we believe that the practical interest of schemas that are intermediate in size between the two extremes that we studied is absolutely real, and we believe that field correlation is a crucial piece of information, we think that managing two relations is the maximum level of complexity that should be inflicted upon the analyst. For this reason, rather than pursuing the avenue of looking for more equivalences, we think that the most promising direction for giving the analyst more choice is *interactive summarization*, that we introduce below. 1.1

Interactive Summarization Since no level of summarization is ideal, it would be useful to implement an interactive workbench where types are first presented in their most summarized form, while the data analyst has the possibility of interactively expanding some internal node. For example, she/he may ask the system to expand a record node collapsed under $\mathcal{K}$-equivalence into a union of many record types, union that is obtained by analyzing the corresponding records according to a finer equivalence. This raises questions about the exact meaning of this request, since our reduction uses the same equivalence all over the dataset, and it is not obvious how to extend it to deal with different equivalences. This also raises questions about the best implementation of this interactive exploration.

Difference Types While $Reduce(T, U, E)$ is a summation operation, there are situations where a difference operation between types would be useful. For example,

in order to compare two different versions of the same dataset, or two different datasets. This is an open research issue.

9 Conclusions

We presented an approach to the problem of the automatic inference of a type for a massive collection of JSON data. The main features of our approach are:

- it can be implemented in a distributed way, hence it can scale over massive data collections;
- it is parametric, hence it can be tuned for different needs;
- it is based on a rigorous formal definition, which enables formal proof of its properties, easy comparison with other approaches, and the design of extensions and variations.

The approach lends itself to different extensions, which we sketched in the paper. As future work we plan to design, study and experimentally analyze some of these extensions.

References

1. `http://webia.lip6.fr/~baazizi/rs/js/vj18`.
2. Apache Spark. `http://spark.apache.org`.
3. The JSON Query Language. `http://www.jsoniq.org`.
4. Json schema definition language. `http://jsoniq.org/docs/JSound/html-single/`.
5. Json schema language. `http://json-schema.org`.
6. Spark dataframe. `https://spark.apache.org/docs/latest/sql-programming-guide.html`.
7. M. A. Baazizi, H. Ben Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani. Schema inference for massive JSON datasets. In *EDBT '17*, 2017.
8. M. A. Baazizi, N. Bidoit, D. Colazzo, N. Malla, and M. Sahakyan. Projection for XML update optimization. In *EDBT '11*, pages 307–318, 2011.
9. M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Counting types for massive JSON datasets. In *DBPL '17*, 2017.
10. V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyên. Type-based xml projection. In *VLDB '06*, pages 271–282, 2006.
11. G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise dtds from XML data. In *VLDB '06*, pages 115–126, 2006.
12. K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.
13. D. Bonetta and M. Brantner. Fad.js: Fast JSON data access using jit-based speculative optimizations. *PVLDB*, 10(12):1778–1789, 2017.
14. P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoc. JSON: data model, query languages and schema specification. In *PODS '17*, pages 123–135, 2017.
15. T. Bray. The javascript object notation (JSON) data interchange format, 2014.
16. S. Cebiric, F. Goasdoué, and I. Manolescu. Query-oriented summarization of RDF graphs. *PVLDB*, 8(12):2012–2015, 2015.
17. R. Ciucanu and S. Staworko. Learning schemas for unordered XML. In *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013), August 30, 2013, Riva del Garda, Trento, Italy.*, 2013.
18. D. Colazzo, G. Ghelli, and C. Sartiani. Typing massive json datasets. In *XLDI '12, Affiliated with ICFP*, 2012.
19. M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. In F. Özcan, G. Koutrika, and S. Madden, editors, *SIGMOD '16*, pages 295–310. ACM, 2016.
20. D. D. Freydenberger and T. Kötzing. Fast learning of restricted regular expressions and dtds. *Theory Comput. Syst.*, 57(4):1114–1158, 2015.
21. M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *SIGMOD '00*, pages 165–176, 2000.
22. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97*, pages 436–445, 1997.
23. T. S. Labs. Studio 3T, 2017. Available at `https://studio3t.com`.
24. Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A fast JSON parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.
25. Z. H. Liu, B. Hammerschmidt, and D. McMahon. Json data management: Supporting schema-less development in RDBMS. In *SIGMOD '14*, pages 1247–1258, 2014.
26. M. Lohrey, S. Maneth, and C. P. Reh. Compression of unordered XML trees. In *ICDT'07*, pages 18:1–18:17, 2017.
27. J. McHugh and J. Widom. Query optimization for xml. In *VLDB '99*, pages 315–326. Morgan Kaufmann Publishers Inc., 1999.

28. M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, Nov. 2005.
29. S. Nestorov, S. Abiteboul, and R. Motwani. Infering structure in semistructured data. *SIGMOD Record*, 26(4):39–43, 1997.
30. S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *SIGMOD '98*, pages 295–306, 1998.
31. F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of json schema. In *WWW '16*, pages 263–273, 2016.
32. S. Scherzinger, E. C. de Almeida, T. Cerqueus, L. B. de Almeida, and P. Holanda. Finding and fixing type mismatches in the evolution of object-nosql mappings. In *Proceedings of the Workshops of the EDBT/ICDT 2016*, 2016.
33. P. Schmidt. mongodb-schema, 2017. Available at `https://github.com/mongodb-js/mongodb-schema`.
34. scrapinghub. Skinfer, 2015. Available at `https://github.com/scrapinghub/skinfer`.
35. L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, and C. Wangz. Schema management for document stores. *Proc. VLDB Endow.*, 8(9):922–933, May 2015.

A Proofs of the properties of *Reduce*

We present here the proofs of the main theorems.

We first introduce a bit of notation that will be used in all the proofs.

Notation A.1 *For any SKER E, and any two E-reduced sets of structural types $\mathcal{M}_1$ and $\mathcal{M}_2$, and for any two sets $\mathcal{F}_1$, $\mathcal{F}_2$ of triples $(k_i, \mathcal{T}_i, \mathtt{q}_i)$, where each $\mathcal{T}_i$ is an E-reduced type, we define the following notation.* 3.8 4.2

$$
\begin{aligned}
\mathcal{M}_1 \setminus_E \mathcal{M}_2 &\triangleq \{\!\!\{\ \mathcal{S}_1 \in \mathcal{M}_1 \mid \nexists \mathcal{S}_2 \in \mathcal{M}_2.\ E(\mathcal{S}_1, \mathcal{S}_2)\ \}\!\!\} \\
\mathcal{M}_1 \cap_E \mathcal{M}_2 &\triangleq \{\!\!\{\ \mathcal{S}_1 \in \mathcal{M}_1 \mid \exists \mathcal{S}_2 \in \mathcal{M}_2.\ E(\mathcal{S}_1, \mathcal{S}_2)\ \}\!\!\} \\
\mathcal{M}_1 \bowtie_E \mathcal{M}_2 &\triangleq \{\!\!\{\ \mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E) \\
&\qquad \mid \mathcal{S}_1 \in \mathcal{M}_1, \mathcal{S}_2 \in \mathcal{M}_2, E(\mathcal{S}_1, \mathcal{S}_2)\ \}\!\!\} \\
\mathcal{F}_1 \setminus_{::} \mathcal{F}_2 &\triangleq \{\!\!\{\ (k_1, \mathcal{T}_1, \mathtt{q}_1) \in \mathcal{F}_1 \\
&\qquad \mid \nexists (k_2, \mathcal{T}_2, \mathtt{q}_2) \in \mathcal{F}_2.\ k_1 = k_2\ \}\!\!\} \\
\mathcal{F}_1 \cap_{::} \mathcal{F}_2 &\triangleq \{\!\!\{\ (k_1, \mathcal{T}_1, \mathtt{q}_1) \in \mathcal{F}_1 \\
&\qquad \mid \exists (k_2, \mathcal{T}_2, \mathtt{q}_2) \in \mathcal{F}_2.\ k_1 = k_2\ \}\!\!\} \\
?(\mathcal{F}) &\triangleq \{\!\!\{\ (k, \mathcal{T}, ?) \mid (k, \mathcal{T}, \mathtt{q}) \in \mathcal{F}\ \}\!\!\} \\
\mathcal{F}_1 \bowtie_{::} \mathcal{F}_2 &\triangleq \{\!\!\{\ (k_1, \mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E), \mathtt{q}_1 \cdot \mathtt{q}_2) \\
&\qquad \mid (k_1, \mathcal{T}_1, \mathtt{q}_1) \in \mathcal{F}_1, (k_1, \mathcal{T}_2, \mathtt{q}_2) \in \mathcal{F}_2\ \}\!\!\}
\end{aligned}
$$

These operators allow us to rewrite the definition of *Reduce* and *Fuse* as follows.

Lemma A.1

$$
\begin{aligned}
&\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E) \\
&\doteq \oplus(\ \circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1\ ) \\
&\mathit{Fuse}(\mathcal{R}_1, \mathcal{R}_2, E) \\
&\doteq \underline{\{}\diamond\mathcal{R}_1 \bowtie_{::} \diamond\mathcal{R}_2 \cup ?(\diamond\mathcal{R}_1 \setminus_{::} \diamond\mathcal{R}_2) \cup ?(\diamond\mathcal{R}_2 \setminus_{::} \diamond\mathcal{R}_1)\underline{\}}
\end{aligned}
$$

Lemma A.2 *For any SKER E, and any two E-reduced types $\mathcal{T}_1$ and $\mathcal{T}_2$, the sets $\circ\mathcal{T}_1 \cap_E \circ\mathcal{T}_2$, $\circ\mathcal{T}_2 \cap_E \circ\mathcal{T}_1$, and $\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2$, are all E-distinct, and, for each pair of them, the E relation defines a bijective function between the two.* 3.8 4.2

Proof The sets $\circ\mathcal{T}_1 \cap_E \circ\mathcal{T}_2$ and $\circ\mathcal{T}_2 \cap_E \circ\mathcal{T}_1$ are E-distinct since each is a subset of a set that is E-distinct. The relation E defines an isomorphism between these two sets: every element of $\circ\mathcal{T}_1 \cap_E \circ\mathcal{T}_2$ E-corresponds to at least one element of $\circ\mathcal{T}_2 \cap_E \circ\mathcal{T}_1$ by construction, and it cannot E-correspond to two of them because, by transitivity, they would be E-equivalent, and the type $\mathcal{T}_2$ would then not be E-reduced. The same holds in the other direction, hence E defines a bijection, and it also defines a bijection between $\circ\mathcal{T}_1 \cap_E \circ\mathcal{T}_2$ and the following set of pairs, mapping every $\mathcal{S}_1$ to the only pair $(\mathcal{S}_1, \mathcal{S}_2)$ where $E(\mathcal{S}_1, \mathcal{S}_2)$:

$$\{\!\!\{\ (\mathcal{S}_1, \mathcal{S}_2) \mid \mathcal{S}_1 \in \circ\mathcal{T}_1, \mathcal{S}_2 \in \circ\mathcal{T}_2, E(\mathcal{S}_1, \mathcal{S}_2)\ \}\!\!\}$$

To every pair of this set, the element $\mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E)$ of $\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2$ corresponds and vice versa. By stability, $\mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E)$ is E-equivalent to both $\mathcal{S}_1$ and $\mathcal{S}_2$, hence we can reason as in the previous case to prove, by transitivity, that no two distinct elements of $\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2$ may be equivalent, hence it is E-reduced, and E is a bijection between it and both of $\circ\mathcal{T}_1 \cap_E \circ\mathcal{T}_2$ and $\circ\mathcal{T}_2 \cap_E \circ\mathcal{T}_1$.

Proof of Lemmas 1 and 2 *The following properties hold.*

1. *For any two E-reduced types $\mathcal{T}_1$, $\mathcal{T}_2$,*
   $\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)$ is E-reduced
2. *For any two E-reduced structural types $\mathcal{S}_1$, $\mathcal{S}_2$,*
   $\mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E)$ is E-reduced

3. *For any* $J, \mathcal{S}$,
$$\vdash^E J : \mathcal{S} \;\Rightarrow\; \mathcal{S} \text{ is } E\text{-reduced}$$
4. *For any* $J_1, \ldots, J_n, \mathcal{T}$,
$$\vdash^E J_1, \ldots, J_n :^{\mathfrak{c}} \mathcal{T} \;\Rightarrow\; \mathcal{T} \text{ is } E\text{-reduced}$$

Proof The first two items are proved my mutual induction. The only interesting case is

$$\begin{array}{l} \mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E) \\ \doteq\ \oplus(\ \circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1\ ) \end{array}$$

The set $\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2$ is E-reduced by Lemma A.2, and $\circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2$ and $\circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1$ are included in $\circ\mathcal{T}_1$ and $\circ\mathcal{T}_2$, which are E-reduced by hypothesis. We have hence just to prove that
4.2 two structural types coming from two different sets among $\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2$, $\circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2$ and $\circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1$ cannot be E-equivalent. If one of them comes from $\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2$ and the other from $\circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2$, they cannot be equivalent since the first is E-isomorphic to $\circ\mathcal{T}_1 \cap_E \circ\mathcal{T}_2$, and elements from $\circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2$ cannot be equivalent to any element of $\circ\mathcal{T}_2$. The same holds for $\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2$ and $\circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1$. Finally, no element of $\circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2$ may be equivalent to one element of $\circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1$ since $\circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2$ only contains types that are not equivalent to any element of $\circ\mathcal{T}_2$.

Properties (3) and (4) follow immediately, since all the union types that are produced by the judgments for $\vdash^E J : \mathcal{S}$ and $\vdash^E J :^{\mathfrak{c}} \mathcal{T}$ are actually produced by a $\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)$ operation applied to arguments that are E-reduced by induction hypothesis.

We can now prove the inclusion theorem.

Theorem 3 (Inclusion)
3.8 *For any* SKER *E and for any two E-reduced types $\mathcal{T}_1$ and $\mathcal{T}_2$:*
$$\mathcal{T}_1 + \mathcal{T}_2 \leq \mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)$$
For any two E-reduced structural types $\mathcal{S}_1$ and $\mathcal{S}_2$:
$$E(\mathcal{S}_1, \mathcal{S}_2) \;\Rightarrow\; \mathcal{S}_1 + \mathcal{S}_2 \leq \mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E)$$

Proof By mutual induction.
We want to prove that:

$$\begin{array}{l} \mathcal{T}_1 + \mathcal{T}_2 \\ \leq\ \oplus(\ \circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1\ ) \end{array}$$

That is:

$$\begin{array}{l} \oplus(\circ(\mathcal{T}_1 + \mathcal{T}_2)) \\ \leq\ \oplus(\ \circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1\ ) \end{array}$$

That is:

$$\begin{array}{l} \mathcal{S} \in (\circ(\mathcal{T}_1 + \mathcal{T}_2)) \Rightarrow \\ [\![\mathcal{S}]\!] \subseteq \bigcup_{\mathcal{S}' \in (\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2\ \cup\ \circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1)} [\![\mathcal{S}']\!] \end{array}$$

4.2 The set $\circ(\mathcal{T}_1 + \mathcal{T}_2)$ can be decomposed as follows.

$$\begin{array}{rl} \circ(\mathcal{T}_1 + \mathcal{T}_2) = & (\circ\mathcal{T}_1 \cap_E \circ\mathcal{T}_2)\ \cup\ (\circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2) \\ & \cup\ (\circ\mathcal{T}_2 \cap_E \circ\mathcal{T}_1)\ \cup\ (\circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1) \end{array}$$

If $\mathcal{S} \in \circ\mathcal{T}_1 \cap_E \circ\mathcal{T}_2$, then there exists $\mathcal{S}_2 \in \circ\mathcal{T}_2$ with $E(\mathcal{S}, \mathcal{S}_2)$ such that $\mathit{Fuse}(\mathcal{S}, \mathcal{S}_2, E)$ belongs to $\circ\mathcal{T}_1 \bowtie_E \circ\mathcal{T}_2$, and, by induction, we know that:

$$E(\mathcal{S}, \mathcal{S}_2) \;\Rightarrow\; [\![\mathcal{S}]\!] \subseteq [\![\mathcal{S} + \mathcal{S}_2]\!] \subseteq [\![\mathit{Fuse}(\mathcal{S}, \mathcal{S}_2, E)]\!]$$

The case for $\mathcal{S} \in \circ\mathcal{T}_2 \cap_E \circ\mathcal{T}_1$ is analogous. The other two cases, $\mathcal{S} \in \circ\mathcal{T}_1 \setminus_E \circ\mathcal{T}_2$ and $\mathcal{S} \in \circ\mathcal{T}_2 \setminus_E \circ\mathcal{T}_1$, are trivial.

We move now to the proof of

$$E(\mathcal{S}_1, \mathcal{S}_2) \;\Rightarrow\; \mathcal{S}_1 + \mathcal{S}_2 \leq \mathit{Fuse}(\mathcal{S}_1, \mathcal{S}_2, E)$$

by cases on the common kind of $\mathcal{S}_1$ and $\mathcal{S}_2$.

If they belong to an atomic kind, the thesis is immediate.

If they are of array type, then we have $\mathcal{S}_1 = [\mathcal{T}_1]$ and $\mathcal{S}_2 = [\mathcal{T}_2]$. We want to prove:

$$\begin{array}{rl} [\![[\mathcal{T}_1]]\!] \cup [\![[\mathcal{T}_2]]\!] \subseteq & [\![\mathit{Fuse}([\mathcal{T}_1], [\mathcal{T}_2], E)]\!] \\ = & [\![[\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)]]\!] \end{array}$$

That is,

$$[\![[\mathcal{T}_1]]\!] \subseteq [\![[\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)]]\!]$$

and

$$[\![[\mathcal{T}_2]]\!] \subseteq [\![[\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)]]\!].$$

Let us prove the first. Assume that $\langle\!\langle V_1, \ldots, V_n \rangle\!\rangle \in [\![[\mathcal{T}_1]]\!]$. This implies that, for any i, we have that $V_i \in [\![\mathcal{T}_1]\!]$.

By induction, $[\![\mathcal{T}_1]\!] \subseteq [\![\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)]\!]$, hence, for any i, we have that $V_i \in [\![\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)]\!]$, hence $\langle\!\langle V_1, \ldots, V_n \rangle\!\rangle \in [\![[\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)]]\!]$.

The inclusion $[\![[\mathcal{T}_2]]\!] \subseteq [\![[\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)]]\!]$ can be proved in the same way.

The last case is that of record types, that is, $\mathcal{S}_1 = \{\!\underline{\ } \diamond\mathcal{S}_1 \underline{\ }\!\}$ and $\mathcal{S}_2 = \{\!\underline{\ } \diamond\mathcal{S}_2 \underline{\ }\!\}$.

We want to prove:

$$[\![\{\!\underline{\ } \diamond\mathcal{S}_1 \underline{\ }\!\}]\!] \cup [\![\{\!\underline{\ } \diamond\mathcal{S}_2 \underline{\ }\!\}]\!] \subseteq [\![\mathit{Fuse}(\{\!\underline{\ } \diamond\mathcal{S}_1 \underline{\ }\!\}, \{\!\underline{\ } \diamond\mathcal{S}_2 \underline{\ }\!\}, E)]\!]$$

We prove the case for $\mathcal{S}_1$, the one for $\mathcal{S}_2$ being analogous.

$$[\![\{\!\underline{\ } \diamond\mathcal{S}_1 \underline{\ }\!\}]\!] \subseteq [\![\mathit{Fuse}(\{\!\underline{\ } \diamond\mathcal{S}_1 \underline{\ }\!\}, \{\!\underline{\ } \diamond\mathcal{S}_2 \underline{\ }\!\}, E)]\!]$$

We rewrite it as follows:

$$\begin{array}{l} [\![\{\!\underline{\ } \mathcal{S}_1 \underline{\ }\!\}]\!] \\ \subseteq\ [\![\{\!\underline{\ } (\diamond\mathcal{S}_1 \bowtie_{::} \diamond\mathcal{S}_2) \cup ?(\diamond\mathcal{S}_1 \setminus_{::} \circ\mathcal{S}_2) \cup ?(\diamond\mathcal{S}_2 \setminus_{::} \circ\mathcal{S}_1)\ \underline{\ }\!\}]\!] \end{array}$$

Consider a record $\mathcal{V} \in [\![\{\!\underline{\ } \mathcal{S}_1 \underline{\ }\!\}]\!]$. By definition,

$$\mathcal{V} = \{\!|\ (k_1, \mathcal{V}_1), \ldots, (k_n, \mathcal{V}_n)\ |\!\}$$

such that:

1. for any $i \in 1...n$, $\exists \mathcal{T}_i, \mathtt{q}_i$ such that $(k_i, \mathcal{T}_i, \mathtt{q}_i)$ belongs to $\diamond\mathcal{S}_1$, and $\mathcal{V}_i \in [\![\mathcal{T}_i]\!]$
2. for any $(k_j, \mathcal{T}_j, !) \in \diamond\mathcal{S}_1$, a pair $(k_j, \mathcal{V}_j)$ is in $\mathcal{V}$.

We want to prove the same properties for $\mathcal{V}$ with respect to

$$\{\!\underline{\ } (\diamond\mathcal{S}_1 \bowtie_{::} \diamond\mathcal{S}_2) \cup ?(\diamond\mathcal{S}_1 \setminus_{::} \circ\mathcal{S}_2) \cup ?(\diamond\mathcal{S}_2 \setminus_{::} \circ\mathcal{S}_1)\ \underline{\ }\!\}$$

We first prove the first property. Assume that the pair 1.19
$(k_i, \mathcal{V}_i)$ belongs to $\mathcal{V}$. By (1) above, we have a triple $(k_i, \mathcal{T}_i, \mathtt{q}_i)$ in $\diamond\mathcal{S}_1$ with $\mathcal{V}_i \in [\![\mathcal{T}_i]\!]$. If a matching k exists in $\mathcal{S}_2$, then we have a triple $(k_i, \mathit{Reduce}(\mathcal{T}_i, \mathcal{T}_2, E), \_)$ in $\diamond\mathcal{S}_1 \bowtie_{::} \diamond\mathcal{S}_2$. By induction, $[\![\mathcal{T}_i]\!] \subseteq [\![\mathit{Reduce}(\mathcal{T}_i, \mathcal{T}_2, E)]\!]$, hence $\mathcal{V}_i \in [\![\mathit{Reduce}(\mathcal{T}_i, \mathcal{T}_2, E)]\!]$, as required. If no matching k exists in $\mathcal{S}_2$, then we have a triple $(k_i, \mathcal{T}_i, ?)$ in $\diamond\mathcal{S}_1 \setminus_{::} \diamond\mathcal{S}_2$, and $\mathcal{V}_i \in [\![\mathcal{T}_i]\!]$ holds by hypothesis.

For the second property, every triple $(k_j, \mathcal{T}_j, !)$ in

$$(\diamond\mathcal{S}_1 \bowtie_{::} \diamond\mathcal{S}_2) \cup ?(\diamond\mathcal{S}_1 \setminus_{::} \circ\mathcal{S}_2) \cup ?(\diamond\mathcal{S}_2 \setminus_{::} \circ\mathcal{S}_1)$$

comes from the $\diamond\mathcal{S}_1 \bowtie_{::} \diamond\mathcal{S}_2$ component and, by definition of $\mathtt{q}_1 \cdot \mathtt{q}_2$, it corresponds to a triple $(k_j, \_, !)$ in $\diamond\mathcal{S}_1$, hence $\mathcal{V}$ contains a field with the key k_j by hypothesis.

We can now prove that the $\mathit{Reduce}(\mathcal{T}_1, \mathcal{T}_2, E)$ operator enjoys the commutativity and associativity properties that enable an efficient distributed map-reduce implementation.

Theorem 4 (Commutativity)

1. *Given two E-reduced types $\mathcal{T}_1$, $\mathcal{T}_2$, we have:*

$$Reduce(\mathcal{T}_1, \mathcal{T}_2, E) \doteq Reduce(\mathcal{T}_2, \mathcal{T}_1, E)$$

2. *Given two structural E-reduced types $\mathcal{S}_1$ and $\mathcal{S}_2$ we have:*

$$E(\mathcal{S}_1, \mathcal{S}_2) \Rightarrow Fuse(\mathcal{S}_1, \mathcal{S}_2, E) \doteq Fuse(\mathcal{S}_2, \mathcal{S}_1, E)$$

Proof Immediate, since the definition is symmetric, modulo order, and E enjoys symmetry.

We need a simple lemma before proving the main theo-
3.22 rem.

Lemma A.3 (Distributivity of join over set union) *For any SKER E, for any E-reduced sets of structural types $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}$, and for any sets $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}$ of triples $(k_i, \mathcal{T}_i, \mathsf{q}_i)$, where each $\mathcal{T}_i$ is an E-reduced type, the following equalities hold.*

$$\begin{array}{lll}(\mathcal{M}_1 \cup \mathcal{M}_2) \bowtie_E \mathcal{M} & = & (\mathcal{M}_1 \bowtie_E \mathcal{M}) \cup (\mathcal{M}_2 \bowtie_E \mathcal{M}) \\ (\mathcal{F}_1 \cup \mathcal{F}_2) \bowtie_{::} \mathcal{F} & = & (\mathcal{F}_1 \bowtie_{::} \mathcal{F}) \cup (\mathcal{F}_2 \bowtie_{::} \mathcal{F}) \\ \mathcal{M} \bowtie_E (\mathcal{M}_1 \cup \mathcal{M}_2) & = & (\mathcal{M} \bowtie_E \mathcal{M}_1) \cup (\mathcal{M} \bowtie_E \mathcal{M}_2) \\ \mathcal{F} \bowtie_{::} (\mathcal{F}_1 \cup \mathcal{F}_2) & = & (\mathcal{F} \bowtie_{::} \mathcal{F}_1) \cup (\mathcal{F} \bowtie_{::} \mathcal{F}_2)\end{array}$$

3.22

Proof By definition of $\bowtie_E$:

$$\begin{array}{l}(\mathcal{M}_1 \cup \mathcal{M}_2) \bowtie_E \mathcal{M} \\ = \{\!\{\ Fuse(\mathcal{S}, \mathcal{S}', E) \mid \mathcal{S} \in \mathcal{M}_1 \cup \mathcal{M}_2,\ \mathcal{S}' \in \mathcal{M}, E(\mathcal{S}, \mathcal{S}')\ \}\!\} \\ = \{\!\{\ Fuse(\mathcal{S}, \mathcal{S}', E) \mid \mathcal{S} \in \mathcal{M}_1,\ \mathcal{S}' \in \mathcal{M}, E(\mathcal{S}, \mathcal{S}')\ \}\!\} \\ \quad \cup \{\!\{\ Fuse(\mathcal{S}, \mathcal{S}', E) \mid \mathcal{S} \in \mathcal{M}_2,\ \mathcal{S}' \in \mathcal{M}, E(\mathcal{S}, \mathcal{S}')\ \}\!\} \\ = (\mathcal{M}_1 \bowtie_E \mathcal{M}) \cup (\mathcal{M}_2 \bowtie_E \mathcal{M})\end{array}$$

By definition of $\bowtie_{::}$:

$$\begin{array}{l}(\mathcal{F}_1 \cup \mathcal{F}_2) \bowtie_{::} \mathcal{F} \\ = \{\!\{\ (k, Reduce(\mathcal{T}, \mathcal{T}', E), \mathsf{q} \cdot \mathsf{q}') \\ \qquad \mid (k, \mathcal{T}, \mathsf{q}) \in (\mathcal{F}_1 \cup \mathcal{F}_2), (k, \mathcal{T}', \mathsf{q}') \in \mathcal{F}\ \}\!\} \\ = \{\!\{\ (k, Reduce(\mathcal{T}, \mathcal{T}', E), \mathsf{q} \cdot \mathsf{q}') \\ \qquad \mid (k, \mathcal{T}, \mathsf{q}) \in \mathcal{F}_1, (k, \mathcal{T}', \mathsf{q}') \in \mathcal{F}\ \}\!\} \\ \quad \cup \{\!\{\ (k, Reduce(\mathcal{T}, \mathcal{T}', E), \mathsf{q} \cdot \mathsf{q}') \\ \qquad \mid (k, \mathcal{T}, \mathsf{q}) \in \cup\mathcal{F}_2, (k, \mathcal{T}', \mathsf{q}') \in \mathcal{F}\ \}\!\} \\ = (\mathcal{F}_1 \bowtie_{::} \mathcal{F}) \cup (\mathcal{F}_2 \bowtie_{::} \mathcal{F})\end{array}$$

The last two cases are analogous.
3.22

Theorem 4 (Associativity)

3.8 *The following two properties hold, for any stable KER E.*

1. *Given three E-reduced types $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$, we have*

$$\begin{array}{l}Reduce(Reduce(\mathcal{T}_1, \mathcal{T}_2, E), \mathcal{T}_3, E) \\ \doteq Reduce(\mathcal{T}_1, Reduce(\mathcal{T}_2, \mathcal{T}_3, E), E)\end{array}$$

2. *Given three E-reduced structural types $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ that are mutually E-equivalent, we have*

$$\begin{array}{l}Fuse(Fuse(\mathcal{S}_1, \mathcal{S}_2, E), \mathcal{S}_3, E) \\ \doteq Fuse(\mathcal{S}_1, Fuse(\mathcal{S}_2, \mathcal{S}_3, E), E)\end{array}$$

Proof We proof (1) and (2) by mutual induction.

We first partition each of $\circ\mathcal{T}_1$, $\circ\mathcal{T}_2$ and $\circ\mathcal{T}_3$ in four parts, that correspond to four possible combinations of $\_ \cap_E \_$ and $\_ \setminus_E \_$, as follows.

$$\begin{array}{lll}M_1^{23} = \{\!\{\ \mathcal{S}_1 \in \circ\mathcal{T}_1 & \mid & \exists \mathcal{S}_2 \in \circ\mathcal{T}_2.\ E(\mathcal{S}_1, \mathcal{S}_2), \\ & & \exists \mathcal{S}_3 \in \circ\mathcal{T}_3.\ E(\mathcal{S}_1, \mathcal{S}_3)\ \}\!\} \\ M_1^{\not{23}} = \{\!\{\ \mathcal{S}_1 \in \circ\mathcal{T}_1 & \mid & \exists \mathcal{S}_2 \in \circ\mathcal{T}_2.\ E(\mathcal{S}_1, \mathcal{S}_2), \\ & & \not\exists \mathcal{S}_3 \in \circ\mathcal{T}_3.\ E(\mathcal{S}_1, \mathcal{S}_3)\ \}\!\} \\ M_1^{\not{23}} = \{\!\{\ \mathcal{S}_1 \in \circ\mathcal{T}_1 & \mid & \not\exists \mathcal{S}_2 \in \circ\mathcal{T}_2.\ E(\mathcal{S}_1, \mathcal{S}_2), \\ & & \exists \mathcal{S}_3 \in \circ\mathcal{T}_3.\ E(\mathcal{S}_1, \mathcal{S}_3)\ \}\!\} \\ M_1^{\not{\not{23}}} = \{\!\{\ \mathcal{S}_1 \in \circ\mathcal{T}_1 & \mid & \not\exists \mathcal{S}_2 \in \circ\mathcal{T}_2.\ E(\mathcal{S}_1, \mathcal{S}_2), \\ & & \not\exists \mathcal{S}_3 \in \circ\mathcal{T}_3.\ E(\mathcal{S}_1, \mathcal{S}_3)\ \}\!\}\end{array}$$

The partitions $\{\!\{\ M_2^{13}, M_2^{\not{13}}, M_2^{\not{13}}, M_2^{\not{\not{13}}}\ \}\!\}$ of $\circ\mathcal{T}_2$ and $\{\!\{\ M_3^{12}, M_3^{\not{12}}, M_3^{\not{12}}, M_3^{\not{\not{12}}}\ \}\!\}$ of $\circ\mathcal{T}_3$ are defined in the same way. Now we can decompose $\circ Reduce(\mathcal{T}_1, \mathcal{T}_2, E)$ as follows. In all of our computations we will make use of distributivity of join over set union (Lemma A.3).

$$\begin{array}{rl}\circ Reduce(\mathcal{T}_1, \mathcal{T}_2, E) = & ((M_1^{23} \cup M_1^{\not{23}}) \bowtie_E (M_2^{13} \cup M_2^{\not{13}})) \\ & \cup M_1^{\not{23}} \cup M_1^{\not{23}} \cup M_2^{\not{13}} \cup M_2^{\not{13}} \\ = & ((M_1^{23} \bowtie_E M_2^{13}) \cup (M_1^{\not{23}} \bowtie_E M_2^{\not{13}})) \\ & \cup M_1^{\not{23}} \cup M_1^{\not{23}} \cup M_2^{\not{13}} \cup M_2^{\not{13}}\end{array}$$

Now we compute $\circ Reduce(Reduce(\mathcal{T}_1, \mathcal{T}_2, E), \mathcal{T}_3, E)$. The first two lines join the components of $\circ Reduce(\mathcal{T}_1, \mathcal{T}_2, E)$ that match some component of $\circ\mathcal{T}_3$ with the corresponding component of $\circ\mathcal{T}_3$, while the last line lists all the non-matching components of $\circ Reduce(\mathcal{T}_1, \mathcal{T}_2, E)$ and $\circ\mathcal{T}_3$.

$$\begin{array}{l}\circ Reduce(Reduce(\mathcal{T}_1, \mathcal{T}_2, E), \mathcal{T}_3, E) = \\ \qquad ((M_1^{23} \bowtie_E M_2^{13}) \bowtie_E M_3^{12}) \\ \qquad \cup (M_1^{\not{23}} \bowtie_E M_3^{\not{12}}) \cup (M_2^{\not{13}} \bowtie_E M_3^{\not{12}}) \\ \qquad \cup (M_1^{\not{23}} \bowtie_E M_2^{\not{13}}) \cup M_1^{\not{23}} \cup M_2^{\not{13}} \cup\ M_3^{\not{12}}\end{array}$$

By reordering the components, we have the following equation for $\circ Reduce(Reduce(\mathcal{T}_1, \mathcal{T}_2, E), \mathcal{T}_3, E)$.

$$\begin{array}{l}\circ Reduce(Reduce(\mathcal{T}_1, \mathcal{T}_2, E), \mathcal{T}_3, E) = \\ ((M_1^{23} \bowtie_E M_2^{13}) \bowtie_E M_3^{12}) \\ \cup (M_1^{23} \bowtie_E M_2^{\not{13}}) \cup (M_1^{\not{23}} \bowtie_E M_3^{\not{12}}) \cup (M_2^{\not{13}} \bowtie_E M_3^{\not{12}}) \\ \cup M_1^{\not{23}} \cup M_2^{\not{13}} \cup M_3^{\not{12}}\end{array}$$

The same computation for $\circ Reduce(\mathcal{T}_1, Reduce(\mathcal{T}_2, \mathcal{T}_3, E), E)$ yields the same result with the only exception of the first term.

$$\begin{array}{l}\circ Reduce(\mathcal{T}_1, Reduce(\mathcal{T}_2, \mathcal{T}_3, E), E) = \\ (M_1^{23} \bowtie_E (M_2^{13} \bowtie_E M_3^{12})) \\ \cup (M_1^{\not{23}} \bowtie_E M_2^{\not{13}}) \cup (M_1^{\not{23}} \bowtie_E M_3^{\not{12}}) \cup\ (M_2^{\not{13}} \bowtie_E M_3^{\not{12}}) \\ \cup M_1^{\not{23}} \cup M_2^{\not{13}} \cup M_3^{\not{12}}\end{array}$$

Hence, we only have to prove that

$$((M_1^{23} \bowtie_E M_2^{13}) \bowtie_E M_3^{12}) = (M_1^{23} \bowtie_E (M_2^{13} \bowtie_E M_3^{12}))$$

By definition, we have the following equalities.

$$\begin{array}{l}((M_1^{23} \bowtie_E M_2^{13}) \bowtie_E M_3^{12}) \\ = \{\!\{\ Fuse(\mathcal{S}_1, \mathcal{S}_2, E) \\ \qquad \mid \mathcal{S}_1 \in M_1^{23}, \mathcal{S}_2 \in M_2^{13},\ E(\mathcal{S}_1, \mathcal{S}_2)\ \}\!\} \bowtie_E M_3^{12} \\ = \{\!\{ Fuse(Fuse(\mathcal{S}_1, \mathcal{S}_2, E), \mathcal{S}_3, E) \\ \qquad \mid \mathcal{S}_1 \in M_1^{23}, \mathcal{S}_2 \in M_2^{13}, \mathcal{S}_3 \in M_3^{12}, \\ \qquad E(\mathcal{S}_1, \mathcal{S}_2), E(Fuse(\mathcal{S}_1, \mathcal{S}_2, E), \mathcal{S}_3)\ \}\!\} \\ (M_1^{23} \bowtie_E (M_2^{13} \bowtie_E M_3^{12})) \\ = \{\!\{ Fuse(\mathcal{S}_1, Fuse(\mathcal{S}_2, \mathcal{S}_3, E), E) \\ \qquad \mid \mathcal{S}_1 \in M_1^{23}, \mathcal{S}_2 \in M_2^{13}, \mathcal{S}_3 \in M_3^{12}, \\ \qquad E(\mathcal{S}_2, \mathcal{S}_3), E(\mathcal{S}_1, Fuse(\mathcal{S}_2, \mathcal{S}_3, E))\ \}\!\}\end{array}$$

By stability, both

$$E(\mathcal{S}_1, \mathcal{S}_2) \wedge E(Fuse(\mathcal{S}_1, \mathcal{S}_2, E), \mathcal{S}_3)$$

and

$$E(\mathcal{S}_2, \mathcal{S}_3) \wedge E(\mathcal{S}_1, Fuse(\mathcal{S}_2, \mathcal{S}_3, E))$$

can be rewritten as

$$E(\mathcal{S}_1, \mathcal{S}_2) \wedge E(\mathcal{S}_2, \mathcal{S}_3),$$

while $Fuse(Fuse(\mathcal{S}_1, \mathcal{S}_2, E), \mathcal{S}_3, E)$ is equivalent to

$$Fuse(\mathcal{S}_1, Fuse(\mathcal{S}_2, \mathcal{S}_3, E), E)$$

by induction, hence we conclude.

(2) Observe that $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ have the same kind, by the hypothesis that they are mutually E-equivalent. We prove (2) by cases on their kind.

If they have an atomic kind, the thesis follows by definition of *Reduce*.

If they are of array type, then we have $\mathcal{S}_1 = [\mathcal{T}_1]$, $\mathcal{S}_2 = [\mathcal{T}_2]$, and $\mathcal{S}_3 = [\mathcal{T}_3]$, for some $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$, and we have:

$$\begin{aligned}
&\mathit{Fuse}(\mathit{Fuse}([\mathcal{T}_1],[\mathcal{T}_2],E),[\mathcal{T}_3],E)\\
&\qquad \doteq \mathit{Fuse}([\mathit{Reduce}(\mathcal{T}_1,\mathcal{T}_2,E)],[\mathcal{T}_3],E)\\
&\qquad \doteq [\mathit{Reduce}(\mathit{Reduce}(\mathcal{T}_1,\mathcal{T}_2,E),\mathcal{T}_3,E)]\\
&\mathit{Fuse}([\mathcal{T}_1],\mathit{Fuse}([\mathcal{T}_2],[\mathcal{T}_3],E),E)\\
&\qquad \doteq \mathit{Fuse}([\mathcal{T}_1],[\mathit{Reduce}(\mathcal{T}_2,\mathcal{T}_3,E)],E)\\
&\qquad \doteq [\mathit{Reduce}(\mathcal{T}_1,\mathit{Reduce}(\mathcal{T}_2,\mathcal{T}_3,E),E)]
\end{aligned}$$

The thesis follows by case (1) and mutual induction.

The last case is that of record types, that is, $\mathcal{S}_1 = \{\!\!\{ \diamond\mathcal{S}_1 \}\!\!\}$, $\mathcal{S}_2 = \{\!\!\{ \diamond\mathcal{S}_2 \}\!\!\}$, and $\mathcal{S}_3 = \{\!\!\{ \diamond\mathcal{S}_3 \}\!\!\}$.

We will follow the same structure as in the proof of the first case, that of $\mathit{Reduce}(\mathit{Reduce}(\mathcal{T}_1,\mathcal{T}_2,E),\mathcal{T}_3,E)$.

As in the first case, we partition $\diamond\mathcal{S}_1$ in four parts F_1^{23}, $F_1^{\not 23}$, $F_1^{2\not 3}$, $F_1^{\not 2\not 3}$, according to the existence of a matching field in $\diamond\mathcal{S}_2$ and of a matching field in $\diamond\mathcal{S}_3$.

$$\begin{aligned}
F_1^{23} &= (\diamond\mathcal{S}_1 \cap_{::} \diamond\mathcal{S}_2) \cap_{::} \diamond\mathcal{S}_3\\
F_1^{\not 23} &= (\diamond\mathcal{S}_1 \cap_{::} \diamond\mathcal{S}_2) \setminus_{::} \diamond\mathcal{S}_3\\
F_1^{2\not 3} &= (\diamond\mathcal{S}_1 \setminus_{::} \diamond\mathcal{S}_2) \cap_{::} \diamond\mathcal{S}_3\\
F_1^{\not 2\not 3} &= (\diamond\mathcal{S}_1 \setminus_{::} \diamond\mathcal{S}_2) \setminus_{::} \diamond\mathcal{S}_3
\end{aligned}$$

3.4 Now we can decompose $\diamond\mathit{Fuse}(\mathcal{S}_1,\mathcal{S}_2,E)$ as follows.

$$\begin{aligned}
\diamond\mathit{Fuse}(\mathcal{S}_1,\mathcal{S}_2,E) &= ((M_1^{23} \cup M_1^{\not 23}) \bowtie_E (M_2^{13} \cup M_2^{1\not 3}))\\
&\quad \cup M_1^{\not 23} \cup M_1^{\not 2\not 3} \cup M_2^{1\not 3} \cup M_2^{13}\\
&= ((M_1^{23} \bowtie_E M_2^{13}) \cup (M_1^{\not 23} \bowtie_E M_2^{1\not 3}))\\
&\quad \cup M_1^{\not 23} \cup M_1^{\not 2\not 3} \cup M_2^{1\not 3} \cup M_2^{13}
\end{aligned}$$

Now we compute $\diamond\mathit{Fuse}(\mathit{Fuse}(\mathcal{S}_1,\mathcal{S}_2,E),\mathcal{S}_3,E)$. The first two lines join the components of $\diamond\mathit{Fuse}(\mathcal{S}_1,\mathcal{S}_2,E)$ that match some component of $\diamond\mathcal{S}_3$ with the corresponding component of $\diamond\mathcal{S}_3$, while the last line lists all the non-matching components of $\diamond\mathit{Fuse}(\mathcal{S}_1,\mathcal{S}_2,E)$ and $\diamond\mathcal{S}_3$.

$$\begin{aligned}
&\diamond\mathit{Fuse}(\mathit{Fuse}(\mathcal{S}_1,\mathcal{S}_2,E),\mathcal{S}_3,E) =\\
&\qquad ((F_1^{23} \bowtie_{::} F_2^{13}) \bowtie_{::} F_3^{12})\\
&\qquad \cup (F_1^{\not 23} \bowtie_{::} F_3^{1\not 2}) \cup (F_2^{1\not 3} \bowtie_{::} F_3^{\not 12})\\
&\qquad \cup (F_1^{2\not 3} \bowtie_{::} F_2^{1\not 3}) \cup F_1^{\not 23} \cup F_2^{1\not 3} \cup F_3^{\not 12}
\end{aligned}$$

By reordering the components, we have the following equation for $\diamond\mathit{Fuse}(\mathit{Fuse}(\mathcal{S}_1,\mathcal{S}_2,E),\mathcal{S}_3,E)$.

$$\begin{aligned}
&\diamond\mathit{Fuse}(\mathit{Fuse}(\mathcal{S}_1,\mathcal{S}_2,E),\mathcal{S}_3,E) =\\
&((F_1^{23} \bowtie_{::} F_2^{13}) \bowtie_{::} F_3^{12})\\
&\cup (F_1^{2\not 3} \bowtie_{::} F_2^{1\not 3}) \cup (F_1^{\not 23} \bowtie_{::} F_3^{1\not 2}) \cup (F_2^{1\not 3} \bowtie_{::} F_3^{\not 12})\\
&\cup F_1^{\not 23} \cup F_2^{1\not 3} \cup F_3^{\not 12}
\end{aligned}$$

3.4 The same computation for $\diamond\mathit{Fuse}(\mathcal{S}_1,\mathit{Fuse}(\mathcal{S}_2,\mathcal{S}_3,E),E)$ yields the same result with the only exception of the first term.

$$\begin{aligned}
&\diamond\mathit{Fuse}(\mathcal{S}_1,\mathit{Fuse}(\mathcal{S}_2,\mathcal{S}_3,E),E) =\\
&(F_1^{23} \bowtie_{::} (F_2^{13} \bowtie_{::} F_3^{12}))\\
&\cup (F_1^{2\not 3} \bowtie_{::} F_2^{1\not 3}) \cup (F_1^{\not 23} \bowtie_{::} F_3^{1\not 2}) \cup (F_2^{1\not 3} \bowtie_{::} F_3^{\not 12})\\
&\cup F_1^{\not 23} \cup F_2^{1\not 3} \cup F_3^{\not 12}
\end{aligned}$$

Hence, we only have to prove that

$$((F_1^{23} \bowtie_{::} F_2^{13}) \bowtie_{::} F_3^{12}) = (F_1^{23} \bowtie_{::} (F_2^{13} \bowtie_{::} F_3^{12}))$$

By definition, we have the following equalities.

$$\begin{aligned}
&((F_1^{23} \bowtie_{::} F_2^{13}) \bowtie_{::} F_3^{12})\\
&= \{\!\!\{ (k, \mathit{Reduce}(\mathcal{T}_1,\mathcal{T}_2,E), \mathtt{q}_1 \cdot \mathtt{q}_2)\\
&\qquad \mid (k,\mathcal{T}_1,\mathtt{q}_1) \in F_1^{23}, (k,\mathcal{T}_2,\mathtt{q}_2) \in F_2^{13} \}\!\!\} \bowtie_{::} F_3^{12}\\
&= \{\!\!\{ (k, \mathit{Reduce}(\mathit{Reduce}(\mathcal{T}_1,\mathcal{T}_2,E),\mathcal{T}_3,E), (\mathtt{q}_1 \cdot \mathtt{q}_2) \cdot \mathtt{q}_3)\\
&\qquad \mid (k,\mathcal{T}_1,\mathtt{q}_1) \in F_1^{23}, (k,\mathcal{T}_2,\mathtt{q}_2) \in F_2^{13},\\
&\qquad (k,\mathcal{T}_3,\mathtt{q}_3) \in F_3^{12} \}\!\!\}\\
&(F_1^{23} \bowtie_{::} (F_2^{13} \bowtie_{::} F_3^{12}))\\
&= \{\!\!\{ (k, \mathit{Reduce}(\mathcal{T}_1, \mathit{Reduce}(\mathcal{T}_2,\mathcal{T}_3,E),E), \mathtt{q}_1 \cdot (\mathtt{q}_2 \cdot \mathtt{q}_3))\\
&\qquad \mid (k,\mathcal{T}_1,\mathtt{q}_1) \in F_1^{23}, (k,\mathcal{T}_2,\mathtt{q}_2) \in F_2^{13},\\
&\qquad (k,\mathcal{T}_3,\mathtt{q}_3) \in F_3^{12} \}\!\!\}
\end{aligned}$$

By induction $\mathit{Reduce}(\mathit{Reduce}(\mathcal{T}_1,\mathcal{T}_2,E),\mathcal{T}_3,E)$ is equivalent to $\mathit{Reduce}(\mathcal{T}_1,\mathit{Reduce}(\mathcal{T}_2,\mathcal{T}_3,E),E)$, associativity of $\mathtt{q}' \cdot \mathtt{q}''$ is immediate, hence we conclude.