Counting Types for Massive JSON Datasets

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

Counting Types for Massive JSON Datasets

Mohamed-Amine Baazizi
Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6
UMR 7606, 4 place Jussieu 75005 Paris
baazizi@ia.lip6.fr

Dario Colazzo
Université Paris-Dauphine, PSL Research University,
CNRS, LAMSADE
75016 PARIS, FRANCE
dario.colazzo@dauphine.fr

Giorgio Ghelli
Università di Pisa
Pisa
ghelli@di.unipi.it

Carlo Sartiani
Università della Basilicata
Potenza
sartiani@gmail.com

ABSTRACT

Type systems express structural information about data, are human readable and hence crucial for understanding code, and are endowed with a formal definition that makes them a fundamental tool when proving program properties. Internal data structures of a database store quantitative information about data, information that is essential for optimization purposes, but is not used for documentation or for correctness proofs. In this paper we propose a new idea: raising a part of the quantitative information from the system-level structures to the type level.

Our proposal is motivated by the problem of schema inference for massive collections of JSON data, which are nowadays often collected from external sources and stored in NoSQL systems without an a-priori schema, which makes a-posteriori schema inference extremely useful. NoSQL systems are oriented towards the management of heterogeneous data, and in this context we claim that quantitative information is important in order to assess the relative weight of different variants.

We propose a type system where the same collection can be described at different levels of abstraction. Different abstraction levels are useful for different purposes, hence we describe a parametric inference mechanism, where a single parameter specifies the chosen trade-off between succinctness and precision for the inferred type. This algorithm is designed for massive JSON collection, and hence admits a simple and efficient map-reduce implementation.

CCS CONCEPTS

•Information systems → Semi-structured data; Data model extensions; •Theory of computation → *Type theory; Logic;*

KEYWORDS

JSON, type systems, schema inference, descriptive schemas, map-reduce

DBPL 2017, Munich, Germany

978-1-4503-5354-0/17/09…$15.00
DOI: 10.1145/3122831.3122837

ACM Reference format:
Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Counting Types for Massive JSON Datasets. In *Proceedings of DBPL 2017, Munich, Germany, September 1, 2017,* 12 pages.
DOI: 10.1145/3122831.3122837

1 INTRODUCTION

Type systems express structural information about data, are human readable and hence crucial for understanding code, and are endowed with a formal definition that makes them a fundamental tool when proving program properties. Internal data structures of a database store quantitative information about data, information that is essential for optimization purposes, but is not used for documentation or for correctness proofs. In this paper we propose a new idea: raising a part of the quantitative information from the system-level structures to the type level.

This idea is motivated by the problem of schema inference for JSON data. The explosion of JSON based NoSQL systems, such as MongoDB or CouchDB, is linked to the need of systems where a priori schema is not required, and with the ability to provide for massive horizontal scaling capability. The lack of a priori schema may be helpful in the first phases of a project, but data cannot be usefully exploited without a knowledge of its structure. When the data is represented in a self-describing formalism, such as JSON, schema design can be usefully replaced by the automatic inference of an a posteriori schema from data. This is not a 'prescriptive' schema, that limits the kind of data that will be stored, but is a 'descriptive' schema, that is extremely useful for the data analyst, who needs to understand the structure of the data in order to get information out of them.

Many type inference algorithms have been defined for this aim, but traditional types are not expressive enough in the context of irregular data, where every type has many variants, and type systems are not able to express the distinction between 'exceptional', 'uncommon', 'common', and 'mandatory' variants, and they are not able to describe the size of collections. Consider, for instance, the following schema (or *type*), inferred from a bibliographical JSON dataset according to the technique described in (Baazizi et al. 2017).

$$[[\{\mathit{title} : \mathtt{Str}, \mathit{text} : (\mathtt{[Str]} + \mathtt{Null}), \mathit{author} : \{\ldots\}?\}]]$$

That schema represents an array of arrays of records. Each record has at most three fields: *title*, *text*, and *author*. The *title* field contains a string, while the *text* field contains either an array of strings

or the special value 'null'. The *author* field contains a record, that is not specified here, and the question mark after its type means that it is optional, while the first two fields are mandatory.

While structural information may suffice in many contexts, there are other applications where a structural description is not sufficient. Indeed, one of the most frequent tasks in big data analysis is data sampling; in this process, it is essential to know how many objects are contained in the dataset, as well as the minimum, maximum, and average lengths of arrays, to correctly dimension the sample size. Another application for which purely structural schemas are not informative enough is the integration of JSON data in existing relational databases (DiScala and Abadi 2016; Liu et al. 2014). In this case, to properly design a target relational schema, it is very important to know the frequency of optional fields in JSON objects, as when several optional fields are *rather infrequent* the schema designer may choose to confine them in a supplementary table, rather than using nullable fields.

1.1 Counting types

In this paper we advocate the use of structural, descriptive schemas enriched with quantitative information, and envision a type system that is able not only to express a type describing the structural properties of a dataset, but that can also enrich this type with information about data distribution. There are many ways to describe data distribution, most of which rely on counting information. Going back to the previous example, we observe that every instance of a type constructor in the type corresponds to a path in the data, and we annotate that type constructor with an absolute value, describing how many items are retrieved by the corresponding path, as shown below (the notation nK is used to abbreviate $n \times 1000$).

$$[\,[\,\{\quad \mathit{title} : \mathtt{Str}^{20K}, \mathit{text} : ([\mathtt{Str}^{19,800K}\,]^{19,800} \oplus \mathtt{Null}^{200})^{20K}, \\ \mathit{author} : \{\ldots\}^{2K}\,\}^{20K}\,]^{20}\,]^{1}$$

Reading right to left, the outermost empty path yields 1 array. The path [*], which we use here to extract the content of an array, yields 20 intermediate arrays, which contain 20K records corresponding to the path [*][*]. The path [*][*].*author* corresponds to 2K records only, which means that most of the 20K records have no author. The path [*][*].*text* yields 20K items, hence the field is present in every record. Of these 20K items, 19,800 are arrays, and 200 are the null value. The path [*][*].*text*[*] yields 19,800K strings, hence each of the 19,800 arrays contains, on average, 1000 strings. Finally, the path [*][*].*title* yields 20K strings.

This information is *cumulative*: we have a total of 20K titles and a total of 2,000 authors, which gives us an idea about how much information we can extract out of this data. It can also be read in a *relative* way by dividing the counter of each field by the counter of its record, the counter of a branch by the total count of the union, or the counter of an array content by the counter of the array. For example, the *title* field has a counter of 20K inside a record with the same counter, which means that this field is mandatory, while the counter of *author* is 2K out of 20K, hence one record out of 10 has an author. In the union type, the first branch counts 19,800 out of 20,000 and the second counts 200 out of 20,000, hence the first branch is taken in 99% of the cases. As for arrays, the content of the intermediate array has a counter of 20K, while the array itself has a counter of 20, which means the 20 internal arrays have an average size of 1,000.

1.2 Bounded size arrays

A different, and more traditional, way of giving information about array size is that of providing bounds for the size, for example in the form of a type [Num 5:20] denoting the type of all arrays that have at least 5 numerical elements and at most 20 elements. This feature is present, for example, in JSON Schema, where one can define "minItems" and "maxItems" properties in an array schema specification (Wright 2016).

Size bounds can be used both in a descriptive type, to summarize the minimal and maximal sizes that have been found in all the arrays in a given position inside a JSON dataset, or in a prescriptive type, to limit a priori the acceptable sizes for a JSON array.

We will show that our cumulative size information about array types and this notion of size bounds can be easily combined into a uniform type system and interact with no problem.

1.3 Union types and field correlation

These quantitative types provide a data description that is extremely informative and succinct but, exactly because it is so succinct, it may also hide some important information. Consider for example the following *author* field.

$$[\,[\,\{\ldots \mathit{author} : \{\ldots, \mathit{address}^{1,000}, \mathit{affil}^{500}\}^{2,000}\,\}^{\cdot\cdot}\,]^{\cdot\cdot}\,]^{1}$$

We know that half authors have an address – 1,000 out of 2,000 – and a quarter of them – 500 – have an affiliation, and that may be enough for some applications. However, we may also wonder about the relation between *address* and *affiliation* and, in particular, whether the presence of one implies or excludes the other, or whether these fields are independent from each other.

Union types allow us to express these different possibilities. Consider for example the following types, where r (meant to represent the *rest* of the record) is, for the sake of simplicity, a mandatory field, hence its index is always equal to that of the encoding:

$$\{\mathit{address}^{1,000}, \mathit{affil}^{500}, r^{2,000}\}^{2,000} \tag{1}$$
$$\{\mathit{address}^{500}, \mathit{affil}^{500}, r^{500}\}^{500} \oplus \{\mathit{address}^{500}, r^{1,500}\}^{1,500} \tag{2}$$
$$\{\mathit{address}^{1,000}, r^{1,000}\}^{1,000} \oplus \{\mathit{affil}^{500}, r^{1,000}\}^{1,000} \tag{3}$$
$$\{\mathit{address}^{1,000}, r^{1,500}\}^{1,500} \oplus \{\mathit{affil}^{500}, r^{500}\}^{500} \tag{4}$$
$$\{\mathit{address}^{1,000}, r^{1,000}\}^{1,000} \oplus \{\mathit{affil}^{500}, r^{500}\}^{500} \oplus \{r^{500}\}^{500} \tag{5}$$

The first type is succinct and gives no correlation information. Indeed, type (1) is a supertype of all those that follow, meaning that, whenever a piece of JSON data satisfies any of the types that follow, it also satisfies (1). Type (2) is more informative and tells us that affiliation implies address: all 500 record with affiliation have an address as well. Type (3) expresses, on the contrary, mutual exclusion between affiliation and address, but is not the only way to express this fact: type (4) expresses the same information. Type (5) expresses again the same information as (3) and (4), but in a way that is, in some sense, canonical, since it separately lists each distinct possibility. When every combination of n optional fields is present, then this complete canonical representation has size 2^n and is the only one that gives a complete representation of the mutual correlation of all fields.

Hence, our approach allows one to give a succinct representation of quantitative information, or allows one to express a more detailed information where correlations are expressed, at the price of a less succinct representation, up to the point of a possible exponential explosion. In what follows we are going to present two different algorithms: one that infers succinct types like that of line (1), and another one that infers detailed types such as that of line (5). We believe that the data analyst may be interested in different trade-offs between succinctness and precision, hence should be given the freedom to choose the one that best fits her needs.

1.4 Contributions

In this paper we present the following contributions:

(1) we define a *counting type system* for JSON data, based on record, array, and union types, where every type constructor is endowed with quantitative information, with a non obvious formal semantics based on sets of multisets;
(2) we define a type inference mechanism that allows one to infer a counting type for a given JSON data collection, based on an associative type-reduction operation, that is parameterized on an equivalence relation;
(3) we present two different instances of the algorithm, based on two different equivalence relations;
(4) we describe a map-reduce implementation of this parametric type inference algorithm and experiment it on JSON datasets collected on the web;
(5) we show that our counting types can be smoothly combined with the standard notion of *bounded size arrays*, and can be still inferred using the same map-reduce approach.

2 RELATED WORK

The study of types expressing quantitative information has been limited, to our knowledge, to the specification of the size of flat collections, typically the size of an array, while never touching the issue of the synthetic description of irregular tree-shaped data. Dependent types and probabilistic types have been used to study the properties of code that deals with quantitative data, but this is orthogonal to our aim, since we do not deal here with type inference for code, but only for data.

The problem of inferring descriptive schema information from datasets has been studied for decades in the programming language and database communities. We only present here the work that is more related to our proposal.

In (Baazizi et al. 2017), by using a type language that can describe the nesting structure of records and arrays, declare optional and mandatory record fields, and represent data variability with the help of union types, we have shown how a type can be inferred for a large JSON dataset via a map-reduce algorithm, by inferring a type for each element and then by merging these types, yielding short times on massive datasets. Our main advance over this approach is the design of the counting type system, as well as a more precise notion of type fusion, that still enjoys the basic property of associativity, and hence the possibility of an efficient map-reduce implementation.

In (Klettke et al. 2015) Klettke et al. deal with the problem of DataGuide inference for JSON data, and with its applications for outlier detection. The mechanism they propose is based on an optimized version of the DataGuide, where each node contains the number of the corresponding nodes. These numbers correspond exactly to the numbers computed by the first of our two inference algorithms, which confirms that our choice is natural. The algorithm they propose for the DataGuide computation is quite different from ours, being based on a depth-first traversal of the structure, while their notion of DataGuide lacks union types and, hence, the possibility of describing data at different levels of abstraction, as well as of expressing correlation properties. But the main difference with that work is the fact that we propose a language-level mechanism, while DataGuides are essentially a system-level notion.

There exist several tools for inferring schemas from JSON data stored inside NoSQL systems.

In (Schmidt 2017) the author describes a JavaScript library that is able to infer, in a streaming fashion, a schema with quantitative information from a MongoDB data collection; this library is quite efficient, but its inference algorithm introduces counting errors when optional fields are found. Studio 3T (Labs 2017) is a commercial front-end for MongoDB that offers a very simple schema inference and analysis feature: while the tool is able to extract fields frequencies, the inference process cannot merge similar types, and the resulting schemas has a huge size, which is comparable to that of the input data.

In (Scherzinger et al. 2016), Scherzinger et al. present an approach for maintaining object-NoSQL mappings. The technique focuses on the detection of mismatches between base types (e.g., Boolean, Integer, String), while the use of other properties is left for future work. A series of proposals for processing JSON datasets have been recently presented with the aim of: efficiently managing JSON repositories with the aid of schema information (Liu et al. 2014; Spoth et al. 2017; Wang et al. 2015), describing a-priori schema information (Pezoa et al. 2016; Wright 2016), and automatically transforming denormalized, nested JSON data into normalized relational data that can be stored and queried into a RDBMS (DiScala and Abadi 2016). However, none of these works deals with large scale inference of a global, descriptive counting schema, and, with the notable exception of (Schmidt 2017), none of these approaches is able to exploit union types to account for data irregularity. Also, none of these approaches presents the ability of parameterizing schema inference to account for the need of different levels of abstraction.

3 COUNTING TYPES

3.1 Syntax and semantics

In our formalization, JSON values are either basic values, records, or arrays. Basic values B include the null value, booleans, numbers n, and strings s, and are represented as follows.

Syntax

$$
\begin{array}{lll}
J ::= & B \mid R \mid A & \\
B ::= & \text{null} \mid \text{true} \mid \text{false} \mid n \mid s & n \in \mathcal{N},\ s \in \mathit{String} \\
R ::= & \{l_1 : J_1, \ldots, l_n : J_n\} & n \geq 0 \\
A ::= & [J_1, \ldots, J_n] & n \geq 0
\end{array}
$$

In our semantics, records represent sets of fields, each field being a key-value pair (l, J), and arrays represent sequences of values. In JSON, a record is well-formed only if all its top-level keys are mutually different; this condition will be checked by the type rules.

Notation 3.1 *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 finite set as $\{\!\{ a_1, \ldots, a_n \}\!\}$, a finite multiset as $\{\!\{ a_1, \ldots, a_n \}\!\}^{\mathrm{m}}$ and a finite list as $\langle\!\langle a_1, \ldots, a_n \rangle\!\rangle$. Empty multiset is $\emptyset^{\mathrm{m}}$ and multiset union is $\cup^{\mathrm{m}}$.*

Semantics

$$
\begin{array}{lcl}
[\![B]\!] & \triangleq & B \\
[\![\{l_1 : J_1, \ldots, l_n : J_n\}]\!] & \triangleq & \{\!\{ (l_1, [\![J_1]\!]), \ldots, (l_n, [\![J_n]\!]) \}\!\} \\
[\![[J_1, \ldots, J_n]]\!] & \triangleq & \langle\!\langle [\![J_1]\!], \ldots, [\![J_n]\!] \rangle\!\rangle
\end{array}
$$

The syntax of counting types is defined below.

$$
\begin{array}{lcll}
\mathbb{T} & ::= & \mathbb{S} \mid \emptyset \mid \mathbb{T} \oplus \mathbb{T} & \text{Types} \\
\mathbb{S} & ::= & \mathbb{B} \mid \mathbb{R} \mid \mathbb{A} & \text{Structural types} \\
\mathbb{B} & ::= & \mathtt{Null}^n \mid \mathtt{Bool}^n \mid \mathtt{Num}^n \mid \mathtt{Str}^n & \text{Basic types} \\
\mathbb{R} & ::= & \{l_1 : \mathbb{T}_1, \ldots, l_j : \mathbb{T}_j\}^n \qquad j \geq 0 & \text{Record types} \\
\mathbb{A} & ::= & [\mathbb{T}]^n & \text{Array types}
\end{array}
$$

The basic features of our counting types is that they represents sets of *multisets* of values, rather than sets of *values*. Hence, while the type $\mathtt{Num}$ denotes the set of values $\mathcal{N} = \{\!\{ 0, 1, \ldots \}\!\}$, the counting type $\mathtt{Num}^3$ denotes the set of three-valued multisets

$$\{\!\{ \{\!\{ n_1, n_2, n_3 \}\!\}^{\mathrm{m}} \mid n_1 \in \mathcal{N}, n_2 \in \mathcal{N}, n_3 \in \mathcal{N} \}\!\}.$$

All the multisets in a counting type have the same size, called the *width* of the type, denoted as $\#(\mathbb{T})$ and computed as follows.

Definition 3.2 (Width of $\mathbb{T}$: $\#(\mathbb{T})$).

$$
\begin{array}{rcl}
\#(\mathtt{Null}^n) = \#(\mathtt{Bool}^n) = \#(\mathtt{Num}^n) = \#(\mathtt{Str}^n) & = & n \\
\#(\{l_1 : \mathbb{T}_1, \ldots, l_j : \mathbb{T}_j\}^n) & = & n \\
\#([\mathbb{T}]^n) & = & n \\
\#(\emptyset) & = & 0 \\
\#(\mathbb{T}_1 \oplus \mathbb{T}_2) & = & \#(\mathbb{T}_1) + \#(\mathbb{T}_2)
\end{array}
$$

To define record and array types we need to first define a couple of path operators, $M/[*]$ and M/l (Definition 3.3), which start from a multiset of arrays or of records, take the indicated step, and flatten out the content using multiset union, yielding again a multiset, so that for example we have the following equalities. Observe that the operators are defined on lists and sets of pairs, which are the semantics counterparts of JSON structures.

$$
\begin{array}{l}
\{\!\{ \langle\!\langle V_1, V_2 \rangle\!\rangle, \langle\!\langle V_2, V_3, V_4 \rangle\!\rangle \}\!\}^{\mathrm{m}}/[*] \;=\; \{\!\{ V_1, V_2, V_2, V_3, V_4 \}\!\}^{\mathrm{m}} \\
\{\!\{ \{\!\{ (l, V_1), (m, V_2) \}\!\}, \{\!\{ (l, V_3), (n, V_4) \}\!\}, \{\!\{ (m, V_5) \}\!\} \}\!\}^{\mathrm{m}}/l \;=\; \{\!\{ V_1, V_3 \}\!\}^{\mathrm{m}}
\end{array}
$$

Definition 3.3 ($V.[*]$, $M/[*]$, $V.l$, M/l).

$$
\begin{array}{lcll}
\langle\!\langle V_1, \ldots, V_n \rangle\!\rangle.[*] & \triangleq & \{\!\{ V_1, \ldots, V_n \}\!\}^{\mathrm{m}} & \\
M/[*] & \triangleq & \cup^{\mathrm{m}}_{V \in M} V.[*] & \\
\{\!\{ (l_1, V_1), \ldots, (l_n, V_n) \}\!\}.l & \triangleq & \{\!\{ V_i \}\!\}^{\mathrm{m}} & \text{if } l = l_i \\
\{\!\{ (l_1, V_1), \ldots, (l_n, V_n) \}\!\}.l & \triangleq & \emptyset^{\mathrm{m}} & \text{if } l \notin \{l_n, \ldots, l_n\} \\
M/l & \triangleq & \cup^{\mathrm{m}}_{V \in M} V.l &
\end{array}
$$

The denotational semantics of counting types is defined as follows.

Definition 3.4 ($[\![\mathbb{T}]\!]$).

Notation

$$MSets^n(S) \quad \triangleq \quad \{\!\{ \{\!\{ a_1, \ldots, a_n \}\!\}^{\mathrm{m}} \mid a_i \in S \}\!\}$$

Domain equations

$$
\begin{array}{lcl}
Recs & = & FiniteSets(Keys \times Values) \\
Arrays & = & Lists(Values) \\
Values & = & BaseValues \cup Recs \cup Arrays
\end{array}
$$

Base Types $\quad Domain : Sets(MSets(BaseValues)))$

$$[\![\mathtt{Num}^k]\!] \quad \triangleq \quad MSets^k(\mathcal{N})$$

$[\![\mathtt{Null}^k]\!], [\![\mathtt{Bool}^k]\!], [\![\mathtt{Str}^k]\!]$: similar

Records $\quad Domain : Sets(MSets(Recs)))$

$$
\begin{array}{l}
[\![\{l_1 : \mathbb{T}_1, \ldots, l_n : \mathbb{T}_n\}^k]\!] \\
\triangleq \{\!\{ M \mid M \in MSets^k(Recs), \forall i \in \{1..n\}.\, M/l_i \in [\![\mathbb{T}_i]\!], \\
\qquad\qquad \forall l \notin \{l_1, \ldots, l_n\}.\, M/l = \emptyset^{\mathrm{m}} \}\!\}
\end{array}
$$

Arrays $\quad Domain : Sets(MSets(Arrays))$

$$[\![[\mathbb{T}]^k]\!] \quad \triangleq \quad \{\!\{ M \mid M \in MSets^k(Arrays),\ (M/[*]) \in [\![\mathbb{T}]\!] \}\!\}$$

Union types $\quad Domain : Sets(MSets(Values))$

$$
\begin{array}{lcl}
[\![\emptyset]\!] & \triangleq & \{\!\{ \emptyset^{\mathrm{m}} \}\!\} \\
[\![\mathbb{T}_1 \oplus \mathbb{T}_2]\!] & \triangleq & \{\!\{ M_1 \cup^{\mathrm{m}} M_2 \mid M_1 \in [\![\mathbb{T}_1]\!],\ M_2 \in [\![\mathbb{T}_2]\!] \}\!\}
\end{array}
$$

Hence, an element of an array type $[\mathbb{T}]^n$ is a multiset of exactly n arrays, where the internal $\mathbb{T}$ describes the multiset union of their contents. For example $[\mathtt{Num}^4]^3$ is the type of a multiset $S = \{\!\{ [1], [], [2, 5, 2] \}\!\}^{\mathrm{m}}$: 3 is the size of S, and 4 is the size of $[\![S]\!]/[*]$.

An element of $[\![\{l_1 : \mathbb{T}_1, \ldots, l_n : \mathbb{T}_n\}^k]\!]$ is a multiset M of k records (sets of pairs) such that the multiset M/l_i of the values associated to each l_i belongs to the corresponding $\mathbb{T}_i$. The multiset M/l may have less than k elements, if the key l is not present in every record of the multiset, or exactly k when the field is mandatory (i.e., if $\#(\mathbb{T}_i) > k$ for some i, then $[\![\{l_1 : \mathbb{T}_1, \ldots, l_n : \mathbb{T}_n\}^k]\!]$ is empty).

For example, $\{l : \mathtt{Num}^1 \oplus \mathtt{Bool}^2, m : [\mathtt{Num}^3]^2\}^3$ is a possible type for the multiset:

$$S = \{\!\{ \{l : 1, m : [1, 3]\},\ \{l : t\},\ \{l : f, m : [1]\} \}\!\}^{\mathrm{m}}.$$

Observe that $[\![S]\!]/l = \{\!\{ 1, t, f \}\!\}^{\mathrm{m}}$ and $[\![S]\!]/m = \{\!\{ \langle\!\langle 1, 3 \rangle\!\rangle, \langle\!\langle 1 \rangle\!\rangle \}\!\}^{\mathrm{m}}$: when all three records are considered, the mandatory l field contains three values ($3 = \#(\mathtt{Num}^1 \oplus \mathtt{Bool}^2)$), while the optional m field only contains two arrays ($2 = \#([\mathtt{Num}^3]^2)$).

Quantities in this type systems are cumulative. Consider the following type, describing a nested structure where every key is mandatory and every array has length 10.

$$[\{a : [\mathtt{Num}^{100}]^{10}, b : \mathtt{Num}^{10}, c : [[\mathtt{Bool}^{1000}]^{100}]^{10}\}^{10}]^1$$

Consider a multiset $\{\!\{ x \}\!\}^{\mathrm{m}}$ belonging to that type. Observe that x is an array, $x[i].c$ is an array and $x[i].c[j]$ is an array, for each i and j less than 10. Although the generic third-level array $x[i].c[j]$ has length 10, the width of the boolean type that describes its content is 1000. The reason is that this internal type does not denote just the content of one array found in a position $x[i].c[j]$, that is, the multiset $\{\!\{ x[i].c[j][k] \mid k < 10 \}\!\}^{\mathrm{m}}$ for some specific choice of i and

j. Rather, $\texttt{Bool}^{1000}$ is the type of the multiset $x/[*]/c/[*]/[*]$ of all values that can be reached through a path $x[i].c[j][k]$, that is, the multiset

$$\{\!\{\, x[i].c[j][k] \mid i < 10,\ j < 10,\ k < 10 \,\}\!\}^{\mathrm{m}}$$

Union types in our system are central, and are not standard. A fundamental feature of our system is the ability to describe the same type at different levels of abstractions. Consider for example, the following multiset with four arrays.

$$\{\!\{ [1], [2,3], [1,1,1,1,1,1,1,1], [t,t] \}\!\}^{\mathrm{m}}$$

It may be fully described by a type such as

$$[\texttt{Int}^1]^1 \oplus [\texttt{Int}^2]^1 \oplus [\texttt{Int}^8]^1 \oplus [\texttt{Bool}^2]^1$$

Or, we may collapse the first two records as in:

$$[\texttt{Int}^3]^2 \oplus [\texttt{Int}^8]^1 \oplus [\texttt{Bool}^2]^1$$

where the first addend describes two arrays whose total content is made of three integers. Or we may collapse everything as in:

$$[\texttt{Int}^{11} \oplus \texttt{Bool}^2]^4$$

which describes four arrays whose total content is two booleans and 11 integers. In a standard type system, a statement $x : \texttt{Int} + \texttt{Bool}$ means that we expect x to be either an integer or a boolean, and we do not know whether it will be one or another. In our type system when we say $M :^{\mathrm{m}} \texttt{Int}^1 \oplus \texttt{Bool}^1$, we mean that each element of the multiset M is either integer or boolean, but we also say that M contains exactly two elements, and we *know* that exactly one is an integer and exactly one is a boolean.

Traditional type systems are designed to prescribe features of unknown data, while ours is designed to describe feature of data that is already known. Traditional union types are needed in order to accommodate for uncertainty, while our union type is used in order to allow one to describe the same data at different levels of abstraction, with different trade-offs between succinctness and precision. For this reason, our union type does not denote set union, but rather a combination of union and product, so that every element of $\texttt{Int}^1 \oplus \texttt{Bool}^1$ is not "either an integer or a boolean", but is a multiset of two elements, one integer *and* one boolean.

Remark 3.5 While this system may look quite strange, it bears a strong relation with the non-counting system, described in (Baazizi et al. 2017). The non-counting system can be obtained by erasing every integer index from a type, and it holds that $J : \mathbb{T}$ implies that J belongs to the erasure of $\mathbb{T}$ in the non-counting system. Type erasure maps our union type $\mathbb{T}_1 \oplus \mathbb{T}_2$ to a standard set-union type. ∎

We define the usual notions of subtyping $\mathbb{T} \leq \mathbb{U}$ and type equivalence $\mathbb{T} \simeq \mathbb{U}$.

Definition 3.6 ($\mathbb{T} \leq \mathbb{U}$, $\mathbb{T} \simeq \mathbb{U}$). A counting type $\mathbb{T}$ is a subtype of $\mathbb{U}$, written $\mathbb{T} \leq \mathbb{U}$, iff $[\![\mathbb{T}]\!] \subseteq [\![\mathbb{U}]\!]$. A counting type $\mathbb{T}$ is equivalent of $\mathbb{U}$, written $\mathbb{T} \simeq \mathbb{U}$, iff $[\![\mathbb{T}]\!] = [\![\mathbb{U}]\!]$.

As usual, a more detailed description that captures a more specific set - in this case a more specific set of multisets - is a subtype of a more general description, as in the examples below.

$$\begin{array}{lrcl} (i) & \{m : Int^1\}^1 \oplus \{l : Int^1\}^1 & \leq & \{m : Int^1, l : Int^1\}^2 \\ (ii) & [Int^2]^1 \oplus [Int^1]^2 & \leq & [Int^3]^3 \end{array}$$

3.2 Inference modulo reduction

In this paper we use *schema* and *type* as synonyms. Schema inference is the process of inferring a schema, which for us is just a type, for a JSON expression. Our system is based on two judgements: $\vdash^E M :^{\mathrm{m}} \mathbb{T}$, that associates a type $\mathbb{T}$ of width n to a multiset M of n elements, and $\vdash^E J : \mathbb{S}$ that associates a structural type $\mathbb{S}$ of width one to the single value J. Our inference algorithm is parametrized with respect to an equivalence relation E which will be discussed later, and is specified in Figure 1.

The type rules of Figure 1 transform atomic values and records into the corresponding structural types in the obvious way. The type inference for the arrays is based on type inference for collection. Elements of a collection are recursively mapped to the corresponding structural types, and all their types are combined together in order to get a common supertype by rule (TypeUnionMultiset), which is the core of the inference algorithm. A more natural rule would have been

$$\frac{\vdash^E M_1 :^{\mathrm{m}} \mathbb{T}_1 \qquad \vdash^E M_2 :^{\mathrm{m}} \mathbb{T}_2}{\vdash^E M_1 \cup^{\mathrm{m}} M_2 :^{\mathrm{m}} \mathbb{T}_1 \oplus \mathbb{T}_2} \text{(BasicTypeUnionMultiset)}$$

but this rule produces a type that is not very helpful for the task of data summarization, since it has one type for each piece of JSON data, hence we generalize $\mathbb{T}_1 \oplus \mathbb{T}_2$ to a parametric function $Reduce(\mathbb{T}_1, \mathbb{T}_2, E)$ that can produce a type that is much more compact. We are going to define this function in the next section, but we anticipate here that it is commutative, associative, that its behavior depends on the E parameter, and that it always yields a supertype of $\mathbb{T} \oplus \mathbb{U}$. This last property ensures soundness of type inference, as specified by the following theorem (the E parameter will be discussed very soon, but its nature is irrelevant for this theorem).

Theorem 3.7. *Assume that $Reduce(\mathbb{T}, \mathbb{U}, E)$ enjoys the following subtype property, on every pair of types $\mathbb{T}$ and $\mathbb{U}$ that can be both generated by type inference:*

$$\forall \mathbb{T}, \mathbb{U}.\ \mathbb{T} \oplus \mathbb{U} \leq Reduce(\mathbb{T}, \mathbb{U}, E)$$

Then, for any JSON value J, and for any multiset M of JSON values:

$$\begin{array}{lcl} \vdash^E J : \mathbb{S} & \Rightarrow & \{\!\{ [\![J]\!] \}\!\}^{\mathrm{m}} \in [\![\mathbb{S}]\!] \\ \vdash^E M :^{\mathrm{m}} \mathbb{T} & \Rightarrow & [\![M]\!] \in [\![\mathbb{T}]\!] \end{array}$$

Type rules suggest an obvious algorithm, where any J and any M is matched with the conclusions of each rule, and the corresponding rule is used to reduce the conclusions to the premises. The rule (TypeUnionMultiset) is non-deterministic, since a multiset may be split in many different ways, and non terminating, in case one of M_1 or M_2 is empty. Non termination is easy to avoid, by never choosing an empty set as M_1 or M_2. Non-determinism is only apparent: since $Reduce(\mathbb{T}, \mathbb{U}, E)$ is associative and commutative, the algorithm may split a multiset in any way that is convenient, which is essential in a map-reduce implementation. This property will be formalized in Theorem 3.18.

In the following sections, we are going to define the $Reduce(\mathbb{T}, \mathbb{U}, E)$ function and to show how its behavior changes depending on the parameter.

$$\frac{}{\vdash^E \texttt{null} : \texttt{Null}^1}\ (\textsc{TypeNull}) \qquad \frac{}{\vdash^E \textbf{true/false} : \texttt{Bool}^1}\ (\textsc{TypeBool}) \qquad \frac{n \in \mathcal{N}}{\vdash^E n : \texttt{Num}^1}\ (\textsc{TypeNumber}) \qquad \frac{s \in \mathit{String}}{\vdash^E s : \texttt{Str}^1}\ (\textsc{TypeString})$$

$$\frac{\forall i.\ \vdash^E J_i : \mathbb{S}_i \quad \forall i,j.\ i \neq j \Rightarrow l_i \neq l_j}{\vdash^E \{l_1 : J_1, \ldots, l_n : J_n\} : \{l_1 : \mathbb{S}_1, \ldots, l_n : \mathbb{S}_n\}^1}\ (\textsc{TypeRec}) \qquad \frac{\vdash^E \{\!|\, J_1, \ldots, J_n \,|\!\}^{\mathrm{m}} :^{\mathrm{m}} \mathbb{T}}{\vdash^E [J_1, \ldots, J_n] : [\mathbb{T}]^1}\ (\textsc{TypeArray})$$

$$\frac{}{\vdash^E \emptyset^{\mathrm{m}} :^{\mathrm{m}} \emptyset}\ (\textsc{TypeEmptyMultiset}) \qquad \frac{\vdash^E J : \mathbb{S}}{\vdash^E \{\!|\, J \,|\!\}^{\mathrm{m}} :^{\mathrm{m}} \mathbb{S}}\ (\textsc{TypeSingleton}) \qquad \frac{\vdash^E M_1 :^{\mathrm{m}} \mathbb{T}_1 \quad \vdash^E M_2 :^{\mathrm{m}} \mathbb{T}_2}{\vdash^E M_1 \cup^{\mathrm{m}} M_2 :^{\mathrm{m}} \mathit{Reduce}(\mathbb{T}_1, \mathbb{T}_2, E)}\ (\textsc{TypeUnionMultiset})$$

Figure 1: Type inference for counting types.

3.3 Parametric reduction

Type *reduction* is based on structural type *merging*. We define *merging* of two *structural* types $\mathbb{S}_1$ and $\mathbb{S}_2$ as the operation of rewriting $\mathbb{S}_1 \oplus \mathbb{S}_2$ into a structural type $\mathbb{S}_3$, that is a type with no outermost union, such that $\mathbb{S}_1 \oplus \mathbb{S}_2 \leq \mathbb{S}_3$, with the aim of getting a type that is more compact even if less precise.

We divide the structural types into six different kinds, as formalized by the following function that maps any $\mathbb{S}$ to an integer:

$$\mathit{kind}(\texttt{Null}^i) = 0 \quad \mathit{kind}(\texttt{Bool}^i) = 1 \quad \mathit{kind}(\texttt{Num}^i) = 2$$
$$\mathit{kind}(\texttt{Str}^i) = 3 \qquad \mathit{kind}(\mathbb{R}) = 4 \qquad \mathit{kind}(\mathbb{A}) = 5$$

It is easy to see that inclusion $\mathbb{S}_1 \oplus \mathbb{S}_2 \leq \mathbb{S}_3$ implies that $\mathbb{S}_1$, $\mathbb{S}_2$ and $\mathbb{S}_3$ have the same kind, that is, two structural types $\mathbb{S}_1$ and $\mathbb{S}_2$ can only be merged if they have the same kind.

Merging base types is easy: a union type $\texttt{Int}^i \oplus \texttt{Int}^j$ can be rewritten as $\texttt{Int}^{i+j}$: both types denote the sets of all multisets that contain exactly $i + j$ integers, hence this merging operation does not entail any loss of type information, and the same holds for the other kinds of base types.

Merging two array types is also quite easy: for any pair $[\mathbb{T}]^i$ and $[\mathbb{U}]^j$, the type $[\mathbb{T} \oplus \mathbb{U}]^{i+j}$ is a structural supertype of $[\mathbb{T}]^i \oplus [\mathbb{U}]^j$. For example, $[\texttt{Int}^3]^1$ and $[\texttt{Int}^2]^2$ can be merged yielding $[\texttt{Int}^3 \oplus \texttt{Int}^2]^{1+2}$, hence $[\texttt{Int}^5]^3$. However, in this case, there is a loss of type information. While $[\texttt{Int}^3]^1 \oplus [\texttt{Int}^2]^2$ specifies that 3 integers are in one array and 2 are somehow split among the other two arrays, the supertype $[\texttt{Int}^5]^3$ describes a set of 3 integer arrays with total content of 5 integer, with no other information. As another example, $[\texttt{Int}^3]^1 \oplus [\texttt{Bool}^3]^1$ describes two homogeneous arrays, but the supertype $[\texttt{Int}^3 \oplus \texttt{Bool}^3]^2$ describes two arrays that collectively contain three integers and three booleans that can be freely mixed. In this case we not only lose information about size of each array, but also about the fact that the two base types are separated.

Finally, if we have two record types such as $\{a : \texttt{Int}^1, b : \texttt{Int}^1\}^1$ and $\{b : \texttt{Bool}^1, c : \texttt{Bool}^1\}^1$, they can be merged by merging the common fields, obtaining

$$\{a : \texttt{Int}^1, b : \texttt{Int}^1 \oplus \texttt{Bool}^1, c : \texttt{Bool}^1\}^2$$

This supertype describes two records, says that a and c are only present in one, but does not specify whether they are both in the same, nor the correlation between the type of b, that is once Int and once Bool, and the presence of the other fields, hence a lot of information is lost by this merging. However, if we substitute

$$\{a : \texttt{Int}^1, b : \texttt{Int}^1\}^1 \oplus \{a : \texttt{Int}^2, b : \texttt{Int}^2\}^2$$

with

$$\{a : \texttt{Int}^3, b : \texttt{Int}^3\}^3$$

we have, this time, no loss of information: both the union type and the merged type denote a multiset of three records, each of them of type $\{a : \texttt{Int}^1, b : \texttt{Int}^1\}^1$. Hence, merging records does not always imply loss of information.

To sum up this *informal* discussion, when two structural types have the same kind, they always admit a structural merge. However, such merge may yield a loss of type information, and the loss is more severe when the merged type are farther one from the other. Hence, we define a reduction operation that is parametrized over a partial equivalence relation (PER) E, and which merges two structural types if and only if they are E-equivalent. In this way, a finer E will merge less pairs, yielding a result that is bigger but more informative. A coarser equivalence will give a different trade-off, since it will merge more pairs, hence producing a result that is more compact but less informative. In the extreme cases, an empty PER will merge nothing, 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.

We need to set up a bit of machinery for this. We first recall what is a PER, and we define the notion of a kind-respecting PER.

Definition 3.8 (Partial Equivalence Relation, kind-respecting PER, $\mathcal{K}$ equivalence). A Partial Equivalence Relation (PER) over a set A is a binary relation on A that is symmetric and transitive.

A Kind-respecting PER (KPER) is a PER E such that

$$E(\mathbb{S}_1, \mathbb{S}_2) \Rightarrow \mathit{kind}(\mathbb{S}_1) = \mathit{kind}(\mathbb{S}_2)$$

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

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

We then define a pair of operators, $\circ\mathbb{T}$ to extract the multiset of the structural *addends* out of a type $\mathbb{T}$, and $\bigoplus M$ to rebuild the original type, so that $\bigoplus(\circ(\mathbb{T}))$ is equivalent to $\mathbb{T}$.

Definition 3.9 ($\circ\mathbb{T}$ (addends of $\mathbb{T}$), $\bigoplus M$). For any type $\mathbb{T}$ and for any multiset M of structural types, the operators $\circ\mathbb{T}$ and $\bigoplus M$ are

defined as follows. The elements of $\circ\mathbb{T}$ are called the *addends* of $\mathbb{T}$.

$$\begin{array}{lcll}
\circ(\mathbb{T}_1 \oplus \mathbb{T}_2) & \triangleq & \circ\mathbb{T}_1 \cup^{\mathfrak{m}} \circ\mathbb{T}_2 & \\
\circ\emptyset & \triangleq & \emptyset^{\mathfrak{m}} & \\
\circ\mathbb{S} & \triangleq & \{\!|\, \mathbb{S} \,|\!\}^{\mathfrak{m}} & \\
\bigoplus(\emptyset^{\mathfrak{m}}) & \triangleq & \emptyset & \\
\bigoplus(\{\!|\, \mathbb{S} \,|\!\}^{\mathfrak{m}}) & \triangleq & \mathbb{S} & \\
\bigoplus(\{\!|\, \mathbb{S} \,|\!\}^{\mathfrak{m}} \cup^{\mathfrak{m}} M) & \triangleq & \mathbb{S} \oplus (\bigoplus M) & \text{if } M \neq \emptyset^{\mathfrak{m}}
\end{array}$$

We also define two operators $\diamond(\mathbb{R})$ to extract the key-field pairs out of $\mathbb{R}$ and $\underline{\{}M\underline{\}}^n$ to rebuild the original type, so that $\underline{\{}\diamond(\mathbb{R})\underline{\}}^n$ is equivalent to $\mathbb{R}$ when $n = \#(\mathbb{R})$.

Definition 3.10 ($\diamond(\mathbb{R})$, $\underline{\{}S\underline{\}}^n$).

$$\begin{array}{lcl}
\diamond(\{l_1 : \mathbb{T}_1, \ldots, l_k : \mathbb{T}_k\}^n) & \triangleq & \{\!|\, (l_1, \mathbb{T}_1), \ldots, (l_k, \mathbb{T}_k) \,|\!\} \\
\underline{\{}\, \{\!|\, (l_1, \mathbb{T}_1), \ldots, (l_k, \mathbb{T}_k) \,|\!\} \,\underline{\}}^n & \triangleq & \{l_1 : \mathbb{T}_1, \ldots, l_k : \mathbb{T}_k\}^n
\end{array}$$

We need a final definition, to give a name to the basic invariant of our algorithm: given a KPER E, whenever two structural E-equivalent types are argument of an union, they will be merged, so that the resulting type is E-reduced, as defined below.

Definition 3.11 (E-reduced). Given any partial equivalence relation E defined on structural types, a type $\mathbb{T}$ is *E-reduced* iff for any union type $\mathbb{T}_1$ that is found at any nesting level inside $\mathbb{T}$, no two distinct addends in $\circ\mathbb{T}_1$ are E-equivalent.

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

Definition 3.12 ($Reduce(\mathbb{T}_1, \mathbb{T}_2, E)$). For any KPER E defined on structural types, and for any two E-reduced types $\mathbb{T}_1$ and $\mathbb{T}_2$, the operator $Reduce(\mathbb{T}_1, \mathbb{T}_2, E)$ is defined as follows.

$$\begin{array}{l}
Reduce(\mathbb{T}_1, \mathbb{T}_2, E) = \\
\quad \bigoplus(\quad \{\!|\, Merge(\mathbb{S}_1, \mathbb{S}_2, E) \mid \mathbb{S}_1 \in \circ\mathbb{T}_1, \mathbb{S}_2 \in \circ\mathbb{T}_2, E(\mathbb{S}_1, \mathbb{S}_2) \,|\!\}^{\mathfrak{m}} \\
\qquad\quad \cup^{\mathfrak{m}} \{\!|\, \mathbb{S}_1 \mid \mathbb{S}_1 \in \circ\mathbb{T}_1,\ \nexists \mathbb{S}_2 \in \circ\mathbb{T}_2.\ E(\mathbb{S}_1, \mathbb{S}_2) \,|\!\}^{\mathfrak{m}} \\
\qquad\quad \cup^{\mathfrak{m}} \{\!|\, \mathbb{S}_2 \mid \mathbb{S}_2 \in \circ\mathbb{T}_2,\ \nexists \mathbb{S}_1 \in \circ\mathbb{T}_1.\ E(\mathbb{S}_1, \mathbb{S}_2) \,|\!\}^{\mathfrak{m}} \quad) \\
Merge(\mathbb{B}^m, \mathbb{B}^n, E) = \mathbb{B}^{m+n} \\
Merge(\mathbb{R}_1, \mathbb{R}_2, E) = \\
\quad \underline{\{}\ \{\!|\, (l, Reduce(\mathbb{T}_1, \mathbb{T}_2, E)) \mid (l, \mathbb{T}_1) \in \diamond(\mathbb{R}_1), (l, \mathbb{T}_2) \in \diamond(\mathbb{R}_2) \,|\!\} \\
\qquad \cup \{\!|\, (l, \mathbb{T}_1) \mid (l, \mathbb{T}_1) \in \diamond(\mathbb{R}_1), \nexists \mathbb{T}_2.\ (l, \mathbb{T}_2) \in \diamond(\mathbb{R}_2) \,|\!\} \\
\qquad \cup \{\!|\, (l, \mathbb{T}_2) \mid (l, \mathbb{T}_2) \in \diamond(\mathbb{R}_2), \nexists \mathbb{T}_1.\ (l, \mathbb{T}_1) \in \diamond(\mathbb{R}_2) \,|\!\} \\
\quad \underline{\}}^{\#(\mathbb{R}_1)+\#(\mathbb{R}_2)} \\
Merge([\mathbb{T}_1]^m, [\mathbb{T}_2]^n, E) = [\ Reduce(\mathbb{T}_1, \mathbb{T}_2, E)\ ]^{m+n}
\end{array}$$

The first line is the most important. Observe that, since both $\mathbb{T}_1$ and $\mathbb{T}_2$ are E-reduced, we have that no addend of $\mathbb{T}_1$ is E-related to another addend of $\mathbb{T}_1$ and the same for $\mathbb{T}_2$. Hence, by transitivity of E, no addend of $\mathbb{T}_1$ can be related to two distinct addends of $\mathbb{T}_2$, and vice versa. Hence, any addend of $\mathbb{T}_1$ is either related to exactly one addend of $\mathbb{T}_2$, hence the two are merged, or it is not related to any, and it goes to the result unchanged. Hence, the first line specifies that all pairs of addends of $\mathbb{T}_1 \oplus \mathbb{T}_2$ that are E-related are merged, while those that have no E-equivalent addend are copied into the result.

The other three lines specify how two equivalent structural types can be merged, and they formalize the description given at the beginning of the section: base types are merged by index addition, array types are merged by index addition and content reduction, and record types use index addition and reduce the type of fields with the same key. These last three lines specify *how* structural types are merged, but it is the E equivalence that specifies *if* they are merged. This will be better discussed later, but please consider the two extreme cases: the empty KPER $\emptyset$, and the maximal KPER $\mathcal{K}$. When E is the empty KPER, then no addend is ever merged with any other addend, hence the last three lines of the definition are useless. When E is the maximal KPER $\mathcal{K}$, whenever two types have the same kind, then they are merged. Any intermediate relation will have a less drastic behavior, and will, for example, specify when two record types, or two array types, should or should not be merged.

3.4 Kind-driven reduction

We are finally ready to instantiate our generic reduction operators.

We start by generalizing the approach of (Baazizi et al. 2017), where two types are merged whenever they have the same kind. This is the coarser equivalence relation that can be used in our type system. Kind-driven reduction $Reduce(\mathbb{T}_1, \mathbb{T}_2, \mathcal{K})$ or simply $\mathcal{K}$-reduction, is based on the idea that whenever two records are met inside a collection, their types are always merged together, the frequency of each key in the two record types is summed and, crucially, the same operation, is propagated down the two types, and the same reduction approach is applied to array types. This yields a very compact type, but some important information may be lost during this process.

Example 3.13. Consider the following two types: $\mathbb{T}_1 = [\{l : \mathbb{U}_1^4\}^4]^2 \oplus \mathsf{Num}^2$ and $T_2 = [\{l : \mathbb{U}_2^2, m : \mathbb{U}_3^2\}^2]^3$, where $\mathbb{U}_i^k$ is a metavariable for a type of width k. The reduction process is described below. *Reduce* invokes *Merge* on structural types that are addends, and *Merge* invokes *Reduce* on the types that are found inside arrays and records. Observe that the l field is mandatory in both of the merged record types – its width is equal to the record type width – and, hence, it is mandatory in the final record type again (the width of $Reduce(\mathbb{U}_1^4, \mathbb{U}_2^2, \mathcal{K})$ is $\#(\mathbb{U}_1^4) + \#(\mathbb{U}_2^2) = 6$). On the other side, the m field is only mandatory in the second record type, but not in the first one, hence in the final record type it only has a width 2 when the record type has width 6, hence it is optional with a frequency of 2/6. In the same way, starting from two array types whose average length is 4/2 and 2/3, we arrive at a merged array type whose average length is (4+2)/(2+3), that is, 6/5. Finally, while in $\mathbb{T}_1$ we have an array type in 2 cases out of 4 and in $\mathbb{T}_2$ we always have an array type, in the merged type we have an array type in 5 cases out of 7. All these statistics are computed by the reduction algorithm and can be read from the final type.

$$\begin{array}{l}
Reduce(\mathbb{T}_1, \mathbb{T}_2, \mathcal{K}) = \\
= Merge([\{l : \mathbb{U}_1^4\}^4]^2, [\{l : \mathbb{U}_2^2, m : \mathbb{U}_3^2\}^2]^3, \mathcal{K}) \oplus \mathsf{Num}^2 \\
= [Reduce(\{l : \mathbb{U}_1^4\}^4, \{l : \mathbb{U}_2^2, m : \mathbb{U}_3^2\}^2, \mathcal{K})]^{(2+3)} \oplus \mathsf{Num}^2 \\
= [Merge(\{l : \mathbb{U}_1^4\}^4, \{l : \mathbb{U}_2^2, m : \mathbb{U}_3^2\}^2, \mathcal{K})]^5 \oplus \mathsf{Num}^2 \\
= [\{l : Reduce(\mathbb{U}_1^4, \mathbb{U}_2^2, \mathcal{K}), m : \mathbb{U}_3^2\}^{(4+2)}]^5 \oplus \mathsf{Num}^2
\end{array}$$

3.5 Key-driven reduction

Kind-driven reduction often yields a good compromise between precision and succinctness. It is precise since, when $\vdash^{\mathcal{K}} J : \mathbb{T}$, every

path such that J/p is not empty has a corresponding path in $\mathbb{T}$, and, for every path in $\mathbb{T}$ that arrives to a type $\mathbb{T}'$ of width $\#(\mathbb{T}')$, the corresponding data exists and is a multiset of $\#(\mathbb{T}')$ elements. However, this description loses interesting information when different record types are mixed. Consider a collection that contains 32 records of type

$$\{a : \mathtt{Int}^1, b : \mathtt{Int}^1, c : \mathtt{Int}^1\}^1$$

and 32 records of type

$$\{d : \mathtt{Int}^1, e : \mathtt{Int}^1, f : \mathtt{Int}^1\}^1.$$

The $\mathcal{K}$-reduction algorithm infers a type

$$\{a : \mathtt{Int}^{32}, b : \mathtt{Int}^{32}, c : \mathtt{Int}^{32}, d : \mathtt{Int}^{32}, e : \mathtt{Int}^{32}, f : \mathtt{Int}^{32}\}^{64}.$$

This type exactly describes the six different vertical paths $/a \ldots /f$ and their results, but says nothing about the correlations among different keys, that is, the fact that the keys are partitioned in two groups, $\{\!|\, a, b, c \,|\!\}$ and $\{\!|\, d, e, f \,|\!\}$, where keys of the same group always appear together. Consider now a situation where we have one record for each of the 64 subsets of $\{\!|\, a, b, c, d, e, f \,|\!\}$. In this case the presences of the different keys have no correlation, but the inferred type is exactly the same.

In some situations the data analyst may be interested in this correlation information, and in this case the following $\mathcal{L}$-equivalence relation may be exploited. It coincides with $\mathcal{K}$-equivalence on any type apart from records. Two record types are $\mathcal{L}$-equivalent iff they have the same sets of 'labels', that is, the same set of keys.

Definition 3.14 ($\mathcal{L}$-equivalence). $\mathcal{L}$-equivalence between pairs of structural types is defined as follows.

$$\begin{array}{rcl} \mathcal{L}([\mathbb{T}_1], [\mathbb{T}_2]) & & \text{always} \\ \mathcal{L}([\mathbb{B}], [\mathbb{B}]) & & \text{always} \\ \mathcal{L}(\mathbb{R}_1, \mathbb{R}_2) & \Leftrightarrow & \mathrm{Keys}(\mathbb{R}_1) = \mathrm{Keys}(\mathbb{R}_2) \end{array}$$

If we consider the first case we described, $\mathcal{K}$-reduction yields the type

$$\{a : \mathtt{Int}^{32}, b : \mathtt{Int}^{32}, c : \mathtt{Int}^{32}, d : \mathtt{Int}^{32}, e : \mathtt{Int}^{32}, f : \mathtt{Int}^{32}\}^{64}$$

while $\mathcal{L}$-reduction yields the type

$$\{a : \mathtt{Int}^{32}, b : \mathtt{Int}^{32}, c : \mathtt{Int}^{32}\}^{32} \\ \oplus \{d : \mathtt{Int}^{32}, e : \mathtt{Int}^{32}, f : \mathtt{Int}^{32}\}^{32}$$

The increase of size is minimal, but the new type is much more informative than the previous. If we consider the second case, however, $\mathcal{L}$-reduction yields a very big type, one with 64 addends:

$$\{\,\}^1 \oplus \{a : \mathtt{Int}^1\}^1 \oplus \{b : \mathtt{Int}^1\}^1 \oplus \{c : \mathtt{Int}^1\}^1 \oplus \{d : \mathtt{Int}^1\}^1 \oplus \ldots \\ \oplus \{a : \mathtt{Int}^1, b : \mathtt{Int}^1\}^1 \oplus \{a : \mathtt{Int}^1, c : \mathtt{Int}^1\}^1 \oplus \ldots$$

In this case, this type is still more informative than the merged type, since it *explicitly* specifies that no correlation exists among the keys. However, its size is comparable with the size of the data, hence its utility as a synthetic description of data is very limited.

In practice, no equivalence is better than the other in general, although $\mathcal{K}$-reduction is better in situations where data is very irregular, hence it is important to be able to produce a description that is very synthetic, while $\mathcal{L}$-reduction is better in situations where few types of records that are quite different are mixed in a single collection, or, more generally, in situations where the correlation between the presence of different fields is important for the analyst. Moreover, $\mathcal{K}$-reduction is better in a first phase when one is interested into a fast preview of the general structure of data, while $\mathcal{L}$-reduction may become more interesting when one wants to get a more detailed view of the structure of data.

3.6 Properties of type inference

Our type inference algorithm enjoys two fundamental properties. The first is soundness, the fact that, independently of the partial equivalence that is chosen for the reduction, the inferred type is always a type of the input collection. The second is associativity and commutativity of type reduction, that ensures that the algorithm can be implemented using a single pass in a map-reduce approach.

By Theorem 3.7, soundness is a consequence of the fact that $\mathbb{T} \oplus \mathbb{U} \leq \mathit{Reduce}(\mathbb{T}, \mathbb{U}, E)$, which we prove now.

Theorem 3.15. *For any KPER E on structural types, for any two E-reduced types $\mathbb{T}_1$ and $\mathbb{T}_2$:*

$$\mathbb{T}_1 \oplus \mathbb{T}_2 \leq \mathit{Reduce}(\mathbb{T}_1, \mathbb{T}_2, E)$$

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

$$E(\mathbb{S}_1, \mathbb{S}_2) \;\Rightarrow\; \mathbb{S}_1 \oplus \mathbb{S}_2 \leq \mathit{Merge}(\mathbb{T}, \mathbb{U}, E)$$

Proof: see Appendix.

Theorem 3.16 (Commutativity). *The following two properties hold.*

(1) *Given two E-reduced types $\mathbb{T}_1, \mathbb{T}_2$, we have*

$$\mathit{Reduce}(\mathbb{T}_1, \mathbb{T}_2, E) \simeq \mathit{Reduce}(\mathbb{T}_2, \mathbb{T}_1, E)$$

(2) *Given two structural E-reduced types $\mathbb{S}_1, \mathbb{S}_2$, we have*

$$E(\mathbb{S}_1, \mathbb{S}_2) \;\Rightarrow\; \mathit{Merge}(\mathbb{S}_1, \mathbb{S}_2, E) \simeq \mathit{Merge}(\mathbb{S}_2, \mathbb{S}_1, E)$$

Theorem 3.17 (Associativity). *The following two properties hold.*

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

$$\mathit{Reduce}(\mathit{Reduce}(\mathbb{T}_1, \mathbb{T}_2, E), \mathbb{T}_3, E) \\ \simeq \mathit{Reduce}(\mathbb{T}_1, \mathit{Reduce}(\mathbb{T}_2, \mathbb{T}_3, E), E)$$

(2) *Given three structural E-reduced types $\mathbb{S}_1$, $\mathbb{S}_2$ and $\mathbb{S}_3$ we have*

$$E(\mathbb{S}_1, \mathbb{S}_2), E(\mathbb{S}_2, \mathbb{S}_3) \;\Rightarrow \\ \mathit{Merge}(\mathit{Merge}(\mathbb{S}_1, \mathbb{S}_2, E), \mathbb{S}_3, E) \\ \simeq \mathit{Merge}(\mathbb{S}_1, \mathit{Merge}(\mathbb{S}_2, \mathbb{S}_3, E), E)$$

Theorem 3.18 (Determinism). *For any JSON value J, and for any multiset M of JSON values, for any KPER E, the following hold:*

$$\vdash^E J : \mathbb{S}_1 \;\text{ and }\; \vdash^E J : \mathbb{S}_2 \quad\Rightarrow\quad [\![\mathbb{S}_1]\!] \simeq [\![\mathbb{S}_2]\!]$$
$$\vdash^E M :^{\mathrm{m}} \mathbb{T}_1 \;\text{ and }\; \vdash^E M :^{\mathrm{m}} \mathbb{T}_2 \quad\Rightarrow\quad [\![\mathbb{T}_1]\!] \simeq [\![\mathbb{T}_2]\!]$$

4 BOUNDED SIZE ARRAYS

A bounded size array type is a type $[T\ i{:}j]$ that specifies the set of all arrays whose size k satisfies the constraint $i \leq k \leq j$. Counting types $[\mathbb{T}^m]^n$ can be usefully combined with bounded-size types, since they give different information. Consider for example the following inferred type.

$$\{\mathit{email} : [\mathtt{Str}^{1,200}\ 0{:}15]^{1,000}, \ldots\}^{1,000}$$

We are collecting records where the email field is an array of strings. The size-bounds part of the type tells us that these arrays range, in length, from 0 - an empty array - to 15. The counting

part of the type tells us that we collected 1,000 such arrays whose average length is 1.2 - hence the length distribution is heavily skewed toward the lower end of the 0-15 range.

Depending on the application, either the bounds or the average length may be the most important piece of information. We are going to define here a system that combines both.

We first present syntax and semantics of bounded size counting array types.

Syntax:

$$\mathbb{A} \quad ::= \quad [\mathbb{T}\ i{:}j]^n \qquad n \in \mathcal{N},\ i \in \mathcal{N},\ j \in (\mathcal{N} \cup \{\infty\})$$

Semantics:

$$[\![\, [\mathbb{T}\ i{:}j]^n \,]\!] \triangleq \{\!|\, M \mid M \in MSets^n(Arrays),\ (M/[*]) \in [\![\mathbb{T}]\!],\ V \in M \ \Rightarrow\ i \le |V.[*]| \le j \,|\!\}$$

A bounded array type $[\mathbb{T}\ i{:}j]^n$ denotes a subset of $[\mathbb{T}\ ]^n$: each element of $[\mathbb{T}\ i{:}j]^n$ is a multiset M of n arrays such that $M/[*]$ belongs to $[\![\mathbb{T}]\!]$, with the further constraint that the size of each single array in M is included between i and j, where $i \in \mathcal{N}$, $j \in (\mathcal{N} \cup \{\infty\})$, and ∞ denotes a value that is greater than every element of $\mathcal{N}$.

For any $M \in [\![[\mathbb{T}\ i{:}j]^n]\!]$, the type $\mathbb{T}$ gives cumulative information about $M/[*]$, while the size-bounds describe each element of M. These two pieces of information are different, but they are related. Consider an array type $[\mathbb{T}\ i{:}j]^n$, where $\#(\mathbb{T}) = m$: this array type denotes a multiset of n arrays whose collective content is composed by m values of type $\mathbb{T}$ and such that each single array has a length that is included between i and j. Since m/n is the average length of each array, we must have that $i \le m/n \le j$, that is, this type is empty when $i > m/n$ or $m/n > j$.

The type inference rule for bounded size arrays is very simple:

$$\frac{\vdash^E \{\!|\, J_1, \ldots, J_n \,|\!\}^{\mathrm{m}} :^{\mathrm{m}} \mathbb{T}}{\vdash^E [J_1, \ldots, J_n] : [\mathbb{T}\ n{:}n]^1} \text{(TypeArray)}$$

Rule (TypeArray) always generates a strict $n{:}n$ bound. A bound $i{:}j$ with $i \neq j$ is only generated when the types of two arrays of different sizes i and j are merged. The merge rule for bounded-size arrays is indeed defined as follows.

$$Merge([\mathbb{T}\ i{:}j]^m, [\mathbb{U}\ k{:}l]^n, E) \triangleq [Reduce(\mathbb{T}, \mathbb{U}, E)\ \min(i,k){:}\max(j,l)]^{m+n}$$

In this way, when many arrays with the same size are merged, we end up with a fixed size type such as $[\mathsf{Num}^{200}\ 2{:}2]^{100}$. When the arrays have different sizes, the resulting type $[\mathbb{T}\ i{:}j]^n$ with $\#(\mathbb{T}) = m$, will specify both the average array size m/n and the minimal and maximal length, respectively i and j, of all the arrays.

The unbounded array types can be described as a degenerate form of bounded size types where the lower bound is always 0 and the upper bound is always '∞'.

Observe that our inference and fusion rules never infer a '∞' upper bound: the inferred upper bound is always just the finite length of the longest array that is found in the corresponding position, so that the '∞' value is only present in our formalization for compatibility with the base case and for the cases when our types are used in a prescriptive way.

Once more, bounded size arrays enjoy the properties of soundness of type inference, and commutativity and associativity of type reduction. They are formalized as in Theorems 3.7, 3.15, 3.16, and are not reported here for space reasons.

5 IMPLEMENTATION AND EXPERIMENTS

In this section we show that the parametric type inference approach effectively takes advantage of distribution to run efficiently on relatively large datasets, and that, more interestingly, it is able to extract valuable structural information.

Implementation and experimental setup. The implementation of our type inference technique uses the *map-reduce* paradigm which enables processing large datasets on a cluster of commodity machines. To asses the feasibility of our approach, we performed our experiments on a cluster of 6 nodes with a modest configuration: each node is equipped with a 2 × 10-cores CPU, 64 GB of RAM and a standard RAID hard-drive, running Apache Spark 1.6.1 and Hadoop File System 2.7 (HDFS). Our technique was implemented in Scala by following the formal specification in the paper and by using an external library to parse the textual representation of JSON objects. Data partitioning for the map phase is left to HDFS, which guarantees a balanced data distribution. Our technique also takes advantage of the local aggregation (*combine*) performed by Spark, thanks to the associativity of the type reduction: the types inferred within each partition are locally reduced before being sent to the *master* node in charge of producing the final result. To measure the gain of using a distributed framework, we have set up two different settings: a centralized setting using one node of the cluster (20 cores and 50GB of RAM), and a distributed one that exploits the full cluster capacity (120 cores and 300GB of RAM).

Dataset	GitHub	Twitter	NYTimes
Statistics about the data			
Size	13 GB	21 GB	21.3 GB
# objects	1,000,001	9,901,087	1,184,943
average textual size	14 KB	2 KB	19 KB
average AST size / height	495.46/4	142.21/3	1238.00 /7
Statistics about the types - execution times			
Kind-driven reduction			
Map phase: avg. type size	495.46	135.44	109.74
Reduce ph.: final type size	655	559	139
Total time (min) - cent./dist.	15/0.7	*Fail/1.3*	19.5/2.8
Key-driven reduction			
Map phase: avg. type size	495.46	135.44	128.54
Reduce ph.: final type size	2,979	2,438	384
Total time (min) - cent./dist.	15.3/0.8	*Fail/3*	20.4/3

Table 1: Experimental results.

Datasets. To evaluate our approach we use three main datasets, that present complementary features. Specifically, the GitHub dataset (GH) does not use arrays and constitutes a good testbed for the measure of field presence; the Twitter dataset (TW), on the contrary, is interesting for the intensive use of arrays; the NYTimes dataset (NYT) allows the same field to have different structures in different instances and is thus useful to study type variability.

These datasets are described in Table 1. The first part of this table reports basic statistics on each collection: its size in GB, the number of its objects, the average size of the textual representation of each of these objects in KB and in terms of the size of the corresponding abstract syntax tree (AST), and the average height of the AST. It can be observed that the NYTimes objects are the largest among the three datasets, which is related to their high level of nesting.

Performance. The total execution times are reported in Table 1. We first observe that both variants of type reduction incur comparable execution times, with the more precise version being slightly slower. Distribution of execution is extremely effective: it enables processing the Twitter dataset (whose processing failed because of the large number of its objects), and it significantly reduces the execution time for GitHub and NYTypes, with factors of 21 and 7.

Succinctness and precision analysis. Both tested equivalence produce types that are quite succinct, and, of course, $\mathcal{K}$-equivalence produces more compact types than $\mathcal{L}$-equivalence. In our table, we measure succinctness by comparing the sizes of the types that we obtain after each phase with the size of the objects. Some form of succinctness is already obtained upon the map phase where each single object is analyzed. 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. This is due to the presence of arrays, which, in this dataset, typically contain 10 objects, whose type is represented by just one type. The succinctness results after reduction are more interesting, and depend on the equivalence used as parameter. If we consider kind equivalence, the size of the resulting type is, for Twitter which represents the worst case, just 4 times bigger than the average size of each of the 9,901,087 individual types of the collection elements. This limited increase means that the element types are very similar, and it indicates that the type that describes the entire collection is quite small, and hence can be reasonably read and understood by the data analyst. If we consider $\mathcal{L}$-equivalence, the size of the resulting type is, for Twitter which again features the worst case, 18 times bigger than the average size of the element types. The counterpart is of course the extraction of a very precise information about field correlations. The type of the Twitter collection contains 55 record types, nested at different levels. The key-driven inferred type shows that 12 of these 55 record types present different key combinations, and it lists all of these combinations, each with its frequency. The same phenomenon is encountered in GitHub and NYTimes where respectively 7 and 8 record types have different shapes. In these cases the increase in size from the kind-driven to the key-driven is more limited. In a typical situation, a data analyst may use the more compact type in order to gain a first understanding of data structure, and may then use the more precise type to get a better understanding.

Counting types analysis. This part of this section is devoted to discussing the benefit of using counting types. This theme is inherently subjective: knowing the frequency of one specific field or the frequency of a combination of fields in a record may be extremely important, or totally irrelevant, depending on the role of the field or the record and on the need of the data analyst. The only way to measure the practical benefit would be by collecting and analyzing interviews to data analysts, which is out of the scope of this paper. Hence, we try here a different approach: we measure the variability of the information that we collect, according to the information-theoretic observation that numbers that are uniform are not very interesting, since they can be easily guessed.

Three features are discussed: the frequency of branches in unions, the frequency of (optional) fields in records, and the cardinality of arrays. For the sake of simplicity, frequencies are grouped into three ranges: marginal which correspond to values below 10%, dominant which correspond to values above 90%, and median which correspond to all values in between. We focus on the distribution of Nulls in 'Null+String' unions, the most common we observed, and the distribution of optional fields. The distribution of Nulls for Github, Twitter and NYTimes datasets, and for each range, are respectively (14, 5, 2), (12, 12, 4) and (5, 1, 2) whereas the distribution of fields for these dataset are (5, 12, 0), (14, 3, 3) and (11, 3, 4). We observe that all datasets present the whole range of situations with fields that are very often null, very often non-null, and fields that are somewhere in between. The same observation holds for field frequencies with an exception for the GitHub dataset where the intermediate case is more common.

The last feature to analyze is the cardinality of arrays. We distinguish between fixed and variable size arrays. We observed that the size of arrays ranges between 0 and 4 when dealing with fixed size arrays, whereas it can range from 0 to 35 in varying size arrays. Many of Twitter arrays have length 2 which, by a deeper examination, revealed that they correspond to longitude and latitude coordinates. The use of fixed size arrays is more common in Twitter than in NYTimes, where variable size arrays are more frequent. By direct observation we noticed that, most of the time, the content of fixed size arrays is a tuple of numeric values, whereas the content of variable size arrays are lists of records.

6 CONCLUSIONS

Types are a fundamental tool to document code and reason about it. They usually express structural information, but we believe they may be usefully extended with quantitative information.

We have presented here a case study for this idea, where a quantitative type system is used in order to describe data. This is useful to assess the size of the data under consideration but is especially useful in a situation where data is somehow irregular, with different variants, since it gives the possibility to specify the frequency of each of the variants.

The basic original features of our type system are the set-of-multisets semantics that departs from the usual set-of-values approach and the peculiar nature of the union type operator. The most original feature of our type-inference algorithm is the parameterization on an equivalence relation, which has practical relevance and has also the advantage of allowing us to prove the basic properties of many different variants of the same algorithm just once.

We have presented the notion of counting types in a very specific setting, that of schema inference. We believe it would be interesting to extend the idea of quantitative types to the more standard scenario where types are used to specify property of unknown data and to specify properties of code, but this is far from easy.

REFERENCES

Mohamed-Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *International Conference on Extending Database Technology (EDBT)*.

Michael DiScala and Daniel J. Abadi. 2016. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data *(SIGMOD '16)*. 295–310. DOI: http://dx.doi.org/10.1145/2882903.2882924

Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2015. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*.

3T Software Labs. 2017. Studio 3T. (2017). Available at `https://studio3t.com`.

Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. 2014. JSON Data Management: Supporting Schema-less Development in RDBMS *(SIGMOD '14)*. 1247–1258. DOI: http://dx.doi.org/10.1145/2588555.2595628

Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema *(WWW '16)*. 263–273. DOI: http://dx.doi.org/10.1145/2872427.2883029

Stefanie Scherzinger, Eduardo Cunha de Almeida, Thomas Cerqueus, Leandro Batista de Almeida, and Pedro Holanda. 2016. Finding and Fixing Type Mismatches in the Evolution of Object-NoSQL Mappings. In *Proceedings of the Workshops of the EDBT/ICDT 2016 (CEUR Workshop Proceedings)*, Themis Palpanas and Kostas Stefanidis (Eds.), Vol. 1558. CEUR-WS.org. http://ceur-ws.org/Vol-1558/paper10.pdf

Peter Schmidt. 2017. mongodb-schema. (2017). Available at `https://github.com/mongodb-js/mongodb-schema`.

William Spoth, Bahareh Sadat Arab, Eric S. Chan, Dieter Gawlick, Adel Ghoneimy, Boris Glavic, Beda Christoph Hammerschmidt, Oliver Kennedy, Seokki Lee, Zhen Hua Liu, Xing Niu, and Ying Yang. 2017. Adaptive Schema Databases. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p84-spoth-cidr17.pdf

Lanjun Wang, Shuo Zhang, Juwei Shi, Limei Jiao, Oktie Hassanzadeh, Jia Zou, and Chen Wangz. 2015. Schema Management for Document Stores. *Proc. VLDB Endow.* 8, 9 (May 2015), 922–933. DOI: http://dx.doi.org/10.14778/2777598.2777601

A. Wright. 2016. *JSON Schema: A Media Type for Describing JSON Documents*. Technical Report. Internet Engineering Task Force.

A APPENDIX

Lemma A.1. *$\oplus$ is monotonic w.r.t subtyping, that is:*

$$T \leq T' \ \wedge \ U \leq U' \ \Rightarrow \ T \oplus U \leq T' \oplus U'$$

Theorem A.2. *For any KPER E on structural types, for any two E-reduced types T_1 and T_2:*

$$T_1 \oplus T_2 \leq \mathit{Reduce}(T_1, T_2, E)$$

For any two E-reduced structural types S_1 and S_2:

$$E(S_1, S_2) \ \Rightarrow \ S_1 \oplus S_2 \leq \mathit{Merge}(T, U, E)$$

Proof.

$$\begin{aligned}
&\mathit{Reduce}(T_1, T_2, E) = \\
&\quad \bigoplus(\ \{\!\{ \ \mathit{Merge}(S_1, S_2, E) \mid S_1 \in \circ T_1, S_2 \in \circ T_2, E(S_1, S_2) \ \}\!\}^{\mathrm{m}} \\
&\qquad \cup^{\mathrm{m}} \{\!\{ \ S_1 \mid S_1 \in \circ T_1, \ \nexists S_2 \in \circ T_2. \ E(S_1, S_2) \ \}\!\}^{\mathrm{m}} \\
&\qquad \cup^{\mathrm{m}} \{\!\{ \ S_2 \mid S_2 \in \circ T_2, \ \nexists S_1 \in \circ T_1. \ E(S_1, S_2) \ \}\!\}^{\mathrm{m}} \quad) \\
&\mathit{Merge}(B^m, B^n, E) = B^{m+n} \\
&\mathit{Merge}(R_1, R_2, E) = \\
&\quad \underline{\{} \ \{\!\{ \ (l, \mathit{Reduce}(T_1, T_2, E)) \mid (l, T_1) \in \diamond(R_1), (l, T_2) \in \diamond(R_2) \ \}\!\} \\
&\qquad \cup \{\!\{ \ (l, T_1) \mid (l, T_1) \in \diamond(R_1), \nexists T_2. \ (l, T_2) \in \diamond(R_2) \ \}\!\} \\
&\qquad \cup \{\!\{ \ (l, T_2) \mid (l, T_2) \in \diamond(R_2), \nexists T_1. \ (l, T_1) \in \diamond(R_2) \ \}\!\} \\
&\quad \underline{\}}^{\#(R_1)+\#(R_2)} \\
&\mathit{Merge}([T_1]^m, [T_2]^n, E) = [\ \mathit{Reduce}(T_1, T_2, E) \ ]^{m+n}
\end{aligned}$$

By mutual induction on size of T_1 and T_2 and, for structural types, by cases. In the induction order, $\mathit{Reduce}(T_1, T_2, E)$ is greater than $\mathit{Merge}(T_1, T_2, E)$, which will be crucial to prove the first property.

First property: No addend of T_1 may be E-equivalent to two distinct addends of T_2 since, by transitivity, these two distinct addends would be E-related, contradicting the hypothesis that T_2 is E-reduced, and the same property holds for T_2. Hence, $T_1 \oplus T_2$ can be rewritten as follows, because every addend of $T_1 \oplus T_2$ appears in the following expression exactly once.

$$\begin{aligned}
\bigoplus(\ & \{\!\{ \ S_1 \oplus S_2 \mid S_1 \in \circ T_1, S_2 \in \circ T_2, E(S_1, S_2) \ \}\!\}^{\mathrm{m}} \\
& \cup^{\mathrm{m}} \{\!\{ \ S_1 \mid S_1 \in \circ T_1, \ \nexists S_2 \in \circ T_2. \ E(S_1, S_2) \ \}\!\}^{\mathrm{m}} \\
& \cup^{\mathrm{m}} \{\!\{ \ S_2 \mid S_2 \in \circ T_2, \ \nexists S_1 \in \circ T_1. \ E(S_1, S_2) \ \}\!\}^{\mathrm{m}} \quad)
\end{aligned}$$

The definition of $\mathit{Reduce}(T_1, T_2, E)$ is obtained by substituting any $S_1 \oplus S_2$ in the first line with $\mathit{Merge}(S_1, S_2, E)$. The result than follows by the inductive hypothesis $S_1 \oplus S_2 \leq \mathit{Merge}(S_1, S_2, E)$ and by monotonicity of union (Lemma A.1).

Second property, by cases on the kind of S_1. Since E is a KPER, $E(S_1, S_2)$ implies that $\mathit{kind}(S_1) = \mathit{kind}(S_2)$.

Kinds 0-3, In this case, S_1 and S_2 can be written as B^n and B^m, and we must prove that $B^n \oplus B^m \leq B^{m+n}$. By definition, every element of $B^n \oplus B^m$ is a multiset of size $m + n$ that only contains values from B. Such a multiset belongs to B^{m+n} by definition.

Kind 5, array types: $S_1 = [T_1']^m$, $S_2 = [T_2']^n$.

We want to prove that

$$S \in [\![[T_1']^m \oplus [T_2']^n]\!] \implies S \in [\![[\mathit{Merge}(T_1', T_2', E)]^{m+n}]\!]$$

that is, $S \in \mathit{MSets}^{m+n}(\mathit{Arrays})$, $(S/[*]\,) \in [\![\mathit{Merge}(T_1', T_2', E)]\!]$.

Assume that $S \in [\![[T_1']^m \oplus [T_2']^n]\!]$. This implies that $S = S_1 \cup^{\mathrm{m}} S_2$ with $S_1 \in [\![[T_1']^m]\!]$ and $S_2 \in [\![[T_2]^n]\!]$. Hence S is a multiset of $m + n$ arrays and $S_1/[*] \in [\![T_1']\!]$ and $S_2/[*] \in [\![T_2']\!]$. By definition of $/[*]$, it holds that $(S_1 \cup^{\mathrm{m}} S_2)/[*] = (S_1/[*] \cup^{\mathrm{m}} S_2/[*]\,)$. From $S_1/[*] \in [\![T_1']\!]$ and $S_2/[*] \in [\![T_2']\!]$, by definition of $[\![\_ \oplus \_]\!]$, we have that

$(S_1/[*] \cup^{\mathrm{m}} S_2/[*]) \in [\![T'_1 \oplus T'_2]\!]$, hence $(S_1 \cup^{\mathrm{m}} S_2)/[*] \in [\![T'_1 \oplus T'_2]\!]$, that is $S/[*] \in [\![T'_1 \oplus T'_2]\!]$, hence $S/[*] \in [\![Reduce(T'_1, T'_2, E)]\!]$ by induction.

Kind 4, record types: $S_1 = \underline{\{}F_1\underline{\}}^m$, $S_2 = \underline{\{}F_2\underline{\}}^n$,

$$
\begin{array}{l}
S_3 = \\
\quad \underline{\{}\ \{\!|\ (l, Reduce(T_1, T_2, E)) \mid (l, T_1) \in \diamond(R_1), (l, T_2) \in \diamond(R_2)\ |\!\} \\
\qquad \cup\ \{\!|\ (l, T_1) \mid (l, T_1) \in \diamond(R_1), \nexists T_2.\ (l, T_2) \in \diamond(R_2)\ |\!\} \\
\qquad \cup\ \{\!|\ (l, T_2) \mid (l, T_2) \in \diamond(R_2), \nexists T_1.\ (l, T_1) \in \diamond(R_2)\ |\!\} \\
\quad \underline{\}}^{m+n}
\end{array}
$$

We split the fields of F_1 in two sets: the first for those common with F_2 and the second for those unique to F_1:

$$F_1 \ = \ \{(l^i_c : T_1^{f'(i)})\}^{i \in 1..k_c} \cup \{(l^i_1 : T_1^{f''(i)})\}^{i \in 1..k_1}$$

and we do the same for F_2, where we split common fields and unique fields in the following sets:

$$F_2 \ = \ \{(l^i_c : T_2^{g'(i)})\}^{i \in 1..k_c} \cup \{(l^i_2 : T_2^{g''(i)})\}^{i \in 1..k_2}$$

Note that, for the common field, we use the same k_c and the same function l^i_c in both cases. In this way, we can now rewrite S_3 as

$$
\begin{array}{l}
\underline{\{}\ \{(l^i_c, Reduce(T_1^{f'(i)}), T_2^{g'(i)}, E)\}^{i \in 1..k_c} \\
\quad \cup\ \{(l^i_1, T_1^{f''(i))})\}^{i \in 1..k_1} \\
\quad \cup\ \{(l^i_2, T_2^{g''(i))})\}^{i \in 1..k_2} \quad \underline{\}}^{m+n}
\end{array}
$$

We want to prove that $S \in [\![\underline{\{}F_1\underline{\}}^m \oplus \underline{\{}F_2\underline{\}}^n]\!]$ implies $S \in [\![S_3]\!]$, that is:

(1) $S \in MSets^{(m+n)}(Recs)$
(2) $\forall i \in 1..k_c.\ S/l^i_c \in [\![Reduce(T_1^{f'(i)}, T_2^{g'(i)}, E)]\!]$
(3) $\forall i \in 1..k_1.\ S/l^i_1 \in [\![T_1^{f''(i)}]\!]$
(4) $\forall i \in 1..k_2.\ S/l^i_2 \in [\![T_2^{g''(i)}]\!]$
(5) $\forall l \notin (\mathrm{Keys}(F_1) \cup \mathrm{Keys}(F_2)).\ S/l = \emptyset^{\mathrm{m}}$

Assume that $S \in [\![\underline{\{}F_1\underline{\}}^m \oplus \underline{\{}F_2\underline{\}}^n]\!]$. This implies that $S = S_1 \cup^{\mathrm{m}} S_2$ with $S_1 \in [\![\underline{\{}F_1\underline{\}}^m]\!]$ and $S_2 \in [\![\underline{\{}F_2\underline{\}}^n]\!]$. Hence S is a multiset of $m+n$ records. We have now to prove the other four properties.

By definition of $\_/l$, we have that, for any key l:

$$S/l \quad = \quad (S_1 \cup^{\mathrm{m}} S_2)/l \quad = \quad S_1/l \ \cup^{\mathrm{m}} S_2/l \qquad (*)$$

Prop. 2: We know by hypothesis that $S_1/l^i_c \in [\![T_1^{f'(i)}]\!]$ and $S_2/l^i_c \in [\![T_2^{f'(i)}]\!]$. Hence, by (*), $S/l^i_c \in [\![T_1^{f'(i)} \oplus T_2^{f'(i)}]\!]$ and, by induction, $S/l^i_c \in [\![Reduce(T_1^{f'(i)}, T_2^{g'(i)}, E)]\!]$.

Prop. 3-4: We know by hypothesis that $S_1/l^i_1 \in [\![T_1^{f''(i)}]\!]$ and that $S_2/l^i_1 = \emptyset^{\mathrm{m}}$. By (*), we conclude that $S/l^i_1 = S_1/l^i_1$, hence $S/l^i_1 \in [\![T_1^{f''(i)}]\!]$. Case 4 is identical.

Prop. 5: Assume that $l \notin (\mathrm{Keys}(F_1) \cup \mathrm{Keys}(F_2))$. Then, $S_1/l = \emptyset^{\mathrm{m}}$ and $S_2/l = \emptyset^{\mathrm{m}}$, hence $S/l = \emptyset^{\mathrm{m}}$.