A Type System for Interactive JSON Schema Inference (Extended Abstract)

Mohamed-Amine Baazizi
Sorbonne Université, CNRS, LIP6 UMR 7606, Paris, France
baazizi@ia.lip6.fr

Dario Colazzo
Université Paris-Dauphine, PSL, LAMSADE, France
dario.colazzo@dauphine.fr

Giorgio Ghelli
Dipartimento di Informatica, Università di Pisa, Italy
ghelli@di.unipi.it

Carlo Sartiani
DIMIE, Università della Basilicata – Potenza, Italy
sartiani@gmail.com

Abstract

In this paper we present the first JSON type system that provides the possibility of inferring a schema by adopting different levels of precision/succinctness for different parts of the dataset, under user control. This feature gives the data analyst the possibility to have detailed schemas for parts of the data of greater interest, while more succinct schema is provided for other parts, and the decision can be changed as many times as needed, in order to explore the schema in a gradual fashion, moving the focus to different parts of the collection, without the need of reprocessing data and by only performing type rewriting operations on the most precise schema.

2012 ACM Subject Classification Theory of computation → Type theory; Information systems → Semi-structured data

Keywords and phrases JSON, type systems, interactive inference

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.101

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version A full version of the paper is available at [3], `https://hal.archives-ouvertes.fr/hal-02112560/file/icalp2019-full.pdf`.

1 Introduction

When a data analyst, or a programmer, accesses a JSON data collection for the first time, it is usually necessary to first have a look at its schema, as it is often the case that the data collection is badly documented and that it requires some visual data navigation in order to be fully understood. To this aim, some automated support can be quite useful, especially when the collection is massive and one is interested in a complete description of the structure.

The problem is that the structure of a data collection can be described at many different levels of abstraction: for example, when a semistructured collection of records is examined, one may either just list which fields are present in at least one element, or one may list every possible field combination. The choice of the abstraction level is extremely important, since an abstract description may hide too much information while, in other cases, a detailed description may be so big as to make the description unreadable. Unfortunately, there is

46th International Colloquium on Automata, Languages, and Programming (ICALP 2019).
Editors: Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi;
Article No. 101; pp. 101:1–101:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

no objective and absolute definition of an "optimal" abstraction level, since it depends on the specific needs of the analyst, who may even need different abstraction levels in different phases of her/his work.

In [4] we proposed a first approach to this problem: a schema inference technique that is parametrized by an Equivalence Relation (ER); through this parameter, the user chooses one specific trade-off between succinctness and precision. This is much better than having no choice, but has important drawbacks. First of all, the choice of the value of the parameter is difficult. Secondly, and more importantly, one is typically very interested in some parts of the collection only, hence there is not a globally optimum trade-off: any intermediate choice between maximal detail and maximal compactness will typically be too compact for the interesting parts of the collection, but too detailed with respect to anywhere else. Hence, we built an interactive workbench, where the analyst can start from an abstract description of the collection, and then ask the system to selectively expand the structural description of some specific parts. Of course, the process can be iterated back and forth across the structural description of the data.

The analyst starts from an initial schema and interactively "expands" and "contracts" different parts of it. The main problem in the design of our workbench was that of understanding the exact meaning of these local "expansion" and "contraction" steps, as well as the lack of formal foundations for the optimization of these operations. In this paper we fill these gaps, by giving the formal foundations of interactive schema inference for massive JSON data. Our first contribution is a non-deterministic type system providing a formal characterisation of schemas featuring different precision levels for different parts of the typed dataset. We then define a deterministic version of this system, that is, a system where the user can choose a precision level through the choice of a *split criterion* parameter, that we will define. We then introduce an explicit representation of proof manipulations, in order to formalize the re-typing process, that is, the process of expanding/contracting a specific point of the schema. In principle, re-typing always involves to re-infer a schema from a portion of the data, which is not feasible in a big data setting. In our implementation, that we describe in [3], we exploit some properties of our *split criteria* in order to perform the re-typing without accessing data for a second time. Hence, we conclude our work by proving the soundness of this implementation technique, and studying the properties of our *split criteria* that allow this optimization.

2 Overview

Consider a massive collection of records, not perfectly homogeneous and sampled by the tiny collection below (left-hand side). The schema inference techniques described in [4] allows one to infer a type for each of the records, so as to obtain the collection of types below (right-hand side).

Data				*Types*			
$\{$	$a : \{j : 0, k : 0\},$	$b : \{bb : 0\}$	$\}$	$\{a : \{j : \texttt{Num}, k : \texttt{Num}\},$	$b : \{bb : \texttt{Num}\}$	$\}$
$\{$	$a : \{j : 0\},$	$c : \{cc : 0\}$	$\}$	$\{a : \{j : \texttt{Num}\},$	$c : \{cc : \texttt{Num}\}$	$\}$
$\{$	$a : \{y : 0, z : 0\}$	$c : \{cd : 0\}$	$\}$	$\{a : \{y : \texttt{Num}, z : \texttt{Num}\},$	$c : \{cd : \texttt{Num}\}$	$\}$
$\{$	$a : \{j : 0\},$	$b : 0$	$\}$	$\{a : \{j : \texttt{Num}\},$	$b : \texttt{Num}$	$\}$

The schema that is synthesized out of this collection of types depends on a parameter that is set by the analyst, which is an equivalence relation (ER) E: the inferred schema is obtained by merging any two types that are E-equivalent. For instance, we may use the

$\mathcal{K}$-equivalence, defined in [4] as the equivalence that equates all pairs of types of the same kind (record, array, or base type). In this way, all record types are merged, thus obtaining the following type, where a question mark indicates that the field is optional.

$$\{a : \{j : \texttt{Num}?, k : \texttt{Num}?, y : \texttt{Num}?, z : \texttt{Num}?\}, b : +_K(\{bb : \texttt{Num}\}, \texttt{Num})?, c : \{cc : \texttt{Num}?, cd : \texttt{Num}?\}?\}$$

Observe that all outer record types are merged, and, inside the record types, the types for the a and c keys are merged. The only two types that are kept distinct are those for the b key, since $\{bb : \texttt{Num}\}$ and $\texttt{Num}$ have two different kinds, hence b has a union type $+_K(\{bb : \texttt{Num}\}, \texttt{Num})$, where the $\mathcal{K}$ labels refers to the equivalence that guided the merging.

Before going on, we now rewrite this schema according to the syntax that we are using in this paper, which is more verbose but a bit more natural for our task. In this syntax, we have a union type in front of every record, array, or base type, so that the type above becomes:

$$\begin{array}{lll} +_K(& \{ & a : +_K(\{j : +_K(\texttt{Num})?, k : +_K(\texttt{Num})?, y : +_K(\texttt{Num})?, z : +_K(\texttt{Num})?\}), \\ & & b : +_K(\{bb : +_K(\texttt{Num})\}, \texttt{Num})?, \\ & & c : +_K(\{cc : +_K(\texttt{Num})?, cd : +_K(\texttt{Num})?\})?\}) \end{array}$$

The more types are merged, the smaller is the resulting schema, and the less precise it results. For example, the above schema is compact but is not very precise, since it does not specify how the different fields are mutually related: it only specifies that a and bb are mandatory, but, for the other fields, any combination is allowed. If the analyst wants more information about field correlation, she/he may move to the $\mathcal{L}$-equivalence, that specifies that two records are equivalent iff the respective sets of top-level field labels are the same, hence merging these equivalent types. In this way, we infer the following type (the $\mathcal{L}$-type):

$$\begin{array}{lll} +_L(& \{ & a : +_L(\quad \{j : +_L(\texttt{Num}), k : +_L(\texttt{Num})\}, \{j : +_L(\texttt{Num})\} \quad), \\ & & b : +_L(\quad \{bb : +_L(\texttt{Num})\}, \texttt{Num}) \\ & \}, & \\ & \{ & a : +_L(\quad \{j : +_L(\texttt{Num})\}, \{y : +_L(\texttt{Num}), z : +_L(\texttt{Num})\} \quad), \\ & & c : +_L(\quad \{cc : +_L(\texttt{Num})\}, \{cd : +_L(\texttt{Num})\} \quad) \\ & \}) & \end{array}$$

This type is much bigger, and it gives a lot of information about label correlation: it shows that b and c are mutually exclusive and one must always be present, and the same for cc and cd. It also shows that j and k are exclusive with y, z, that the presence of y, z inside a implies the presence of c, and so on.

The ability to start from a maximally compact type to get an overview of the schema, and then to move to a schema that is extremely detailed is already interesting but, in practice, the expanded schema may be overwhelming and may produce a lot of repetition. The analyst is often interested in just a subset of the input data and would like to be able to study the structure of that subset. For example, in this case the analyst may be mostly interested in the a field, and would like to be able to drill down on its structure only. Hence, she/he would like to start from the $\mathcal{K}$-type and to expand the a field only:

$$\begin{array}{lll} +_K(\{ & a : +_L(& \{j : +_L(\texttt{Num}), k : +_L(\texttt{Num})\}, \{j : +_L(\texttt{Num})\}, \{y : +_L(\texttt{Num}), z : +_L(\texttt{Num})\} \quad), \\ & b : +_K(& \{bb : +_K(\texttt{Num})\}, +_K(\texttt{Num}) \quad)?, \\ & c : +_K(& \{cc : +_K(\texttt{Num})?, cd : +_K(\texttt{Num})?\} \quad)? \ \} \quad) \end{array}$$

In this schema, we have full information about the type of the a field, while the outer structure and the c field are still represented in the compact style of the $\mathcal{K}$-types. Of course the analyst should be free now to also expand the c field, obtaining the following type.

$$+_K(\ \{\ a : +_L(\quad \{j : +_L(\texttt{Num}), k : +_L(\texttt{Num})\}, \{j : +_L(\texttt{Num})\}, \{y : +_L(\texttt{Num}), z : +_L(\texttt{Num})\}\quad),$$
$$\qquad b : +_K(\quad \{bb : +_K(\texttt{Num})\}, +_K(\texttt{Num})\quad)?,$$
$$\qquad c : +_L(\quad \{cc : +_L(\texttt{Num})\}, \{cd : +_L(\texttt{Num})\}\quad)?\ \}\quad)$$

Finally, we may collapse the a field, getting the following type.

$$+_K(\ \{\ a : +_K(\quad \{j : +_K(\texttt{Num})?, k : +_K(\texttt{Num})?, y : +_K(\texttt{Num})?, z : +_K(\texttt{Num})?\}\quad),$$
$$\qquad b : +_K(\quad \{bb : +_K(\texttt{Num})\}, +_K(\texttt{Num})\quad)?,$$
$$\qquad c : +_L(\quad \{cc : +_L(\texttt{Num})\}, \{cd : +_L(\texttt{Num})\}\quad)?\ \}\quad)$$

The formal definition of this operation of *local-equivalence-switch* is less obvious than it may appear. Essentially, the basic idea is that we apply different type-inference techniques to different parts of the data, but these parts are scattered in the dataset: in our example, the a fields are interleaved with the b and c fields. We would like to describe each step in the expand/collapse structure as a rewrite operation in the typing proof-tree, but this rewrite is non-local on the data, which is not trivial to formalize. Moreover, we would like that the relationship between a schema and the described data were fully described by the type, with its $\mathcal{K}$ and $\mathcal{L}$ annotations, and did not depend on the sequence of steps that has been used to arrive to that type. This property is very important for the analyst, who will usually not even remember the sequence of steps that she/he has used in order to arrive at the current schema, but still would like to have an interpretation of the meaning of that schema. The formalization of these properties is the theme of the paper.

3 Syntax and Semantics

We represent JSON values through the following grammar, that corresponds to actual JSON syntax, with the only exception that we do not put quotes around the keys (the labels) of the records. Following [6], we assume key uniqueness in records.

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**

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, J), and arrays represent sequences of values. We will use J to range over JSON expressions and $\mathbb{J}$ to range over lists (or *collections*) of JSON expressions.

▶ **Notation 1.** *We use $L_1 \oplus L_2$ for list concatenation and $e.L$ to add e at the beginning of a list L. We use* collection *as a synonym for* list*, in situations where the order is there for technical reasons but is otherwise irrelevant.*

In the full paper [3] we define a semantic function $[\![\, J \,]\!]$ that maps each JSON term J to the corresponding mathematical entity, e.g. a number term into a number, a record into a set of pairs, an array into a list. That semantics is standard. As an example, the semantics of records is defined as $[\![\, \{l_1 : J_1, \ldots, l_n : J_n\} \,]\!] \triangleq \{\ (l_1, [\![\, J_1 \,]\!]), \ldots, (l_n, [\![\, J_n \,]\!])\ \}$

Our JSON types obey the following grammar. Every *union type* $\mathcal{T}$ is a union $+_{\mathcal{C}}(\mathcal{S}_1, \ldots, \mathcal{S}_n)$ of *structural types* $\mathcal{S}_1, \ldots, \mathcal{S}_n$, where the optional $\mathcal{C}$ subscript on non zero-ary union types can be ignored for the moment, and later will be used to indicate the "split criterion" that has been used in order to infer that part of the type. Every structural type

is either a base type $\mathcal{B}$, a record type $\mathcal{R}$ or an array type $\mathcal{A}$; record types and array types are defined in terms of union types. Each record field in a record type is labeled with a quantifier indicating whether the field is optional, noted as ?, or mandatory, noted as !. In our examples we will often omit the "!" symbol and only keep "?" for optional fields.

Syntax

$\mathcal{U}$	::=	$+(\mathcal{S}_1,\ldots,\mathcal{S}_n) \mid +_{\mathcal{C}}(\mathcal{S}_1,\ldots,\mathcal{S}_{n+1})$	$n \geq 0$	**Union Types**
$\mathcal{S}$	::=	$\mathcal{B} \mid \mathcal{R} \mid \mathcal{A}$		**Struct. types**
$\mathcal{B}$	::=	$\mathtt{Null} \mid \mathtt{Bool} \mid \mathtt{Num} \mid \mathtt{Str}$		**Basic types**
$\mathcal{R}$	::=	$\{l_1 : \mathcal{U}_1\mathtt{q}_1,\ldots,l_n : \mathcal{U}_n\mathtt{q}_n\}$	$n \geq 0\ , \mathtt{q}_i \in \{\ ?,!\ \}$	**Record types**
$\mathcal{A}$	::=	$[\mathcal{U}]$		**Array types**

The formal semantics of types is standard, and needs some explanation only in the records case. A record is a set of pairs, hence the empty record type { } denotes a singleton that only contains the empty record, that is, the empty set of pairs. A one-field record type is a set of singletons when the field is mandatory, and it also contains the empty record when the field is optional. Finally, a record that contains n fields is just a set that contains n pairs, hence a record that belongs to $\{l_1 : \mathcal{U}_1\mathtt{q}_1,\ldots,l_n : \mathcal{U}_n\mathtt{q}_n\}$ is just the union of one record for each field in the type. When a field $\mathcal{U}_i\mathtt{q}_i$ is optional, then the corresponding field in the value may be missing, which is captured by the presence of the empty set in the semantics of that field. In the semantics of $+(\mathcal{S}_1,\ldots,\mathcal{S}_n)$ we adopt the usual convention that $\cup_{i\in 1..0}\mathcal{S}_i$ is the empty set, hence $[\![+(\ )]\!] = \emptyset$. In this phase we just ignore the $\mathcal{C}$ annotation. The semantics of base types is standard, and we omit it in this extended abstract.

Semantics

$$
\begin{array}{lcl}
[\![\{\ \}]\!] & \triangleq & \{\ \emptyset\ \} \\
[\![\{l : \mathcal{U}!\}]\!] & \triangleq & \{\ \{\ (l, V)\ \} \mid V \in [\![\mathcal{U}]\!]\ \} \\
[\![\{l : \mathcal{U}?\}]\!] & \triangleq & [\![\{l : \mathcal{U}!\}]\!] \cup [\![\{\ \}]\!] \\
[\![\{l_1 : \mathcal{U}_1\mathtt{q}_1,\ldots,l_n : \mathcal{U}_n\mathtt{q}_n\}]\!] & \triangleq & \{\ R_1 \cup \ldots \cup R_n \mid R_i \in [\![\{l_i : \mathcal{U}_i\mathtt{q}_i\}]\!],\ i = 1,\ldots,n\ \}
\end{array}
$$

$$[\![[\mathcal{U}]]\!] \triangleq \{\ (V_1,\ldots,V_n) \mid n \geq 0,\ V_i \in [\![\mathcal{U}]\!]\ \}$$

$$[\![+_{[\mathcal{C}]}(\mathcal{S}_1,\ldots,\mathcal{S}_n)]\!] \triangleq \bigcup_{i\in 1..n} [\![\mathcal{S}_i]\!]$$

We partition the structural types into six kinds $\{\ \mathtt{Null}, \mathtt{Bool}, \mathtt{Num}, \mathtt{Str}, \{\ \}, [\ ]\ \}$ by using the following *kind*() function, that returns the *kind* of each structural type. $kind(K) = K$ for $K \in \{\ \mathtt{Null}, \mathtt{Bool}, \mathtt{Num}, \mathtt{Str}\ \}$ while $kind(\mathcal{A}) = [\ ]$ and $kind(\mathcal{R}) = \{\ \}$. We also define a *kind*() function that maps every JSON term to its kind – we omit the obvious definition. We say that a collection $\mathbb{J}$ of JSON terms is *kind-homogeneous* if it is not empty and all its elements have the same kind, in which case $kind(\mathbb{J})$ denotes that kind; $kind(\mathbb{J})$ is undefined when $\mathbb{J}$ is empty or when $\mathbb{J}$ is not kind-homogeneous.

4 Type System

4.1 The non-deterministic type system

In [4] we introduced a parametric deterministic type system, that is, a function that, given a JSON collection $\mathbb{J}$ and a parameter E, returns a type $\mathcal{U}$ for $\mathbb{J}$. The user could influence the type by choosing a specific equivalence relation for E: a finer equivalence results into a type that is more informative but bigger, while a coarser equivalence yields a type that is more compact but less informative. We focused our attention on two equivalence relations only, which we called $\mathcal{L}$ and $\mathcal{K}$, hence the analyst could choose, for each data collection, between two types, the $\mathcal{L}$-type, big and detailed, or the $\mathcal{K}$-type, smaller but less informative.

That approach is not flexible enough. The analyst may prefer to use a different size-precision trade-off for different parts of the data, depending on her/his interests. Hence, we need to formalize the notion that one term collection may have many different types, hence we need a type system that is much more flexible than the one in [4].

This inference system is based on two judgments that are mutually recursive: $\vdash \mathbb{J} :_u \mathcal{U}$ and $\vdash \mathbb{J} :_s \mathcal{S}$. Type inference rules are shown in Figure 1. The judgment $\vdash \mathbb{J} :_u \mathcal{U}$ specifies that the union type $\mathcal{U}$ describes the collection $\mathbb{J}$. Rule (EMPTY) specifies that the empty collection has the empty union type, while rule (+) splits $\mathbb{J}$, non-deterministically, into n kind-homogeneous subcollections $(\mathbb{J}_1, \ldots, \mathbb{J}_n)$. The *split criterion* $\mathcal{C}$ describes how $\mathbb{J}$ has been split. It is a partial function that describes how to split a collection of JSON terms. In this type system it is chosen non-deterministically, it may be different in any instance of the rule, can be optionally reported in the type, and it is only used to record how one specific collection has been split. In the deterministic variant of the system, presented in the next subsection, it will be chosen in a systematic way. The judgment $\vdash \mathbb{J} :_s \mathcal{S}$ assigns a structural type $\mathcal{S}$ to the kind-homogeneous collection $\mathbb{J}$, so that $kind(\mathbb{J}) = kind(\mathcal{S})$.

▶ **Definition 2** (*split*($\mathbb{J}$), split criterion $\mathcal{C}$). *For any non-empty collection of JSON terms, split*($\mathbb{J}$) *is the set of all partitionings of* $\mathbb{J}$ *into kind-homogeneous subcollections (abbreviated k-hom). A* split criterion $\mathcal{C}$ *is a partial function such that, for any non-empty collection of JSON terms* $\mathbb{J}$ *that belongs to its domain,* $\mathcal{C}(\mathbb{J}) \in split(\mathbb{J})$.

$$split(\mathbb{J}) = \{ (\mathbb{J}_1, \ldots, \mathbb{J}_n) \mid \mathbb{J}_1 \oplus \ldots \oplus \mathbb{J}_n = \mathbb{J}, \ \forall i, j.\ i \neq j \Rightarrow \mathbb{J}_i \cap \mathbb{J}_j = \emptyset, \forall i \in 1..n.\ \mathbb{J}_i \text{ is k-hom} \}$$

The three rules of Figure 1 for structural types can only be applied to kind-homogeneous collections. The notations used in the structural rules are introduced below.

$$\text{(EMPTY)}\ \frac{}{\vdash \emptyset :_u +()} \qquad \text{(+)}\ \frac{\mathcal{C} \text{ is a split criterion} \quad (\mathbb{J}_1, \ldots, \mathbb{J}_n) = \mathcal{C}(\mathbb{J}) \quad \forall i = 1, \ldots, n.\ \vdash \mathbb{J}_i :_s \mathcal{S}_i}{\vdash \mathbb{J} :_u +(\mathcal{S}_1, \ldots, \mathcal{S}_n) \text{ or } +_{\mathcal{C}}(\mathcal{S}_1, \ldots, \mathcal{S}_n)}$$

$$\text{(BASE)}\ \frac{kind(\mathbb{J}) = \mathcal{B}}{\vdash \mathbb{J} :_s \mathcal{B}} \qquad \text{(REC)}\ \frac{kind(\mathbb{J}) = \{\ \} \quad \text{Let } (a_1, \ldots, a_n) = keys(\mathbb{J}) \quad \forall i = 1, \ldots, n.\ \vdash \mathbb{J}/a_i :_u \mathcal{U}_i \quad q_i = Q(\frac{|\mathbb{J}/a_i|}{|\mathbb{J}|})}{\vdash \mathbb{J} :_s \{a_1 : \mathcal{U}_1 q_1, \ldots, a_n : \mathcal{U}_n q_n\}}$$

$$\text{(ARRAY)}\ \frac{kind(\mathbb{J}) = [\,] \quad \vdash \mathbb{J}/[*] :_u \mathcal{U}}{\vdash \mathbb{J} :_s [\mathcal{U}]}$$

Figure 1 Type inference rules.

Rule (+) is the key rule. Consider again the collection of Section 2.

$$\begin{array}{llllll} (& \{ & a : \{j : 0, k : 0\}, & b : \{bb : 0\} & \}, & \{ \quad a : \{j : 0\}, \quad c : \{cc : 0\} \quad \} \\ & \{ & a : \{y : 0, z : 0\} & c : \{cd : 0\} & \}, & \{ \quad a : \{j : 0\}, \quad b : 0 \qquad\quad \} \quad) \end{array}$$

If we split it into two different subsets, one for each different choice of the top-level labels, we will get a union type with two addends, similar to the $\mathcal{L}$-type presented in Section 2. However, if we consider the trivial split where all records are put together, we will have a type with only one top level addend, such as the three types whose root is labeled with $+_K$ in Section 2. We could also decide to split the collection in three subsets, in which case the resulting type will be the union of three types, one for each subset. A finer split will result into a union type that is bigger but where each addend is more precise. A coarser split will

yield a union type with less addends, and where each addend will be less precise. Every application of a union rule can use a different approach to splitting, and every application of a union rule corresponds to a different + in the resulting type. Hence, this rule gives us the flexibility that we need in order to model the process of interactive type modification.

For the rule (REC), we first introduce some notations.

▶ **Notation 3** ($J.a$, $\mathbb{J}/a, Q(x)$, $keys(\mathbb{J})$).

$$\begin{array}{llll} \{\ldots, a : J, \ldots\}.a = (\,J\,) & & Q(x) = \,! \;\; \textit{if } x = 1 \\ \{a_1 : J_1, \ldots, a_n : J_n\}.a = (\,) & \textit{if } \forall i \in 1..n.\ a_i \neq a & Q(x) = \,? \;\; \textit{if } 0 < x < 1 \\ \mathbb{J}/a = \bigoplus_{J \in \mathbb{J}} J.a & & keys(\mathbb{J}) = \bigcup_{J \in \mathbb{J}} (keys(J)) \end{array}$$

The rule (REC) specifies that, in order to analyze a kind-homogeneous collection of records, which may have different structures, we extract the collection of all keys that appear in any of them. For each key a_i, we collect the content of that key in all records and we infer a union type $\mathcal{U}_i$ for the collection $\mathbb{J}/a_i$. $\mathcal{U}_i$ will be the type for a_i in the result. Its qualifier will depend on the fraction of records where the a_i field is present, and will be ! if and only if the fraction is 1. In $Q(\,|\mathbb{J}/a_i| \,/\, |\mathbb{J}|\,)$, $|C|$ denotes the cardinality of a collection C.

The rule (ARRAY) is based on the $\mathbb{J}/[*]$ operator, that returns the content of all arrays in $\mathbb{J}$ as defined below; observe that both $(\,)/[*]$ and $(\,[\,], \ldots, [\,]\,)/[*]$ return the empty collection.

▶ **Notation 4** ($\mathbb{J}/[*]$). *For any collection of arrays, the operator $\mathbb{J}/[*]$ is defined as follows.*

$$(\,[J_1^1, \ldots, J_{n_1}^1], \ldots, [J_1^m, \ldots, J_{n_m}^m]\,)/[*] \triangleq (\,J_1^1, \ldots, J_{n_1}^1, \ldots, J_1^m, \ldots, J_{n_m}^m\,)$$

The use of the $\mathbb{J}/a$ and $\mathbb{J}/[*]$ operators in the typing rules is a technical contribution of this work and, in combination with the approach of typing a whole collection at the same time, is the fundamental tool that allows us to express the fact that the same criterion will be used for data that is not consecutive in the dataset but is related by the fact that it is reached through the same path.

This type system enjoys soundness, in the following sense.

▶ **Theorem 5** (Soundness). *For any JSON collection $\mathbb{J}$, union type $\mathcal{U}$, structural type $\mathcal{S}$:*

$$\vdash \mathbb{J} :_u \mathcal{U} \;\Rightarrow\; [\![\mathbb{J}]\!] \subseteq [\![\mathcal{U}]\!] \qquad \vdash \mathbb{J} :_s \mathcal{S} \;\Rightarrow\; [\![\mathbb{J}]\!] \subseteq [\![\mathcal{S}]\!]$$

▶ **Example 6.** Consider a collection $\mathbb{J} = (\,20, [1,3,5], [\,], [1, \text{true}], [2,4]\,)$. We want to find a union type $\mathcal{U}$ such that $\vdash \mathbb{J} :_u \mathcal{U}$. We start with the (+) rule, which divides $\mathbb{J}$ into kind-homogeneous subsets to be separately analyzed. For example, it may be divided into four homogeneous collections $(\,20\,), (\,[1,3,5], [2,4]\,), (\,[\,]\,), (\,[1, \text{true}]\,)$. The (BASE) rule assigns `Num` to $(\,20\,)$. The (ARRAY) rule reduces the problem $\vdash (\,[1,3,5], [2,4]\,) :_s \mathcal{S}_1$ to $\vdash (\,1,3,5,2,4\,) :_u \mathcal{U}_1$. If the split criterion keeps all the integers together, the (BASE) rule assigns `Num` to the integers, hence we have $\vdash (\,1,3,5,2,4\,) :_u +(\texttt{Num})$, hence $\vdash (\,[1,3,5], [2,4]\,) :_s [+(\texttt{Num})]$.

In this way, starting from a four-way split of the original collection, we prove the following judgment: $\vdash \mathbb{J} :_u +(\texttt{Num}, [+(\texttt{Num})], [+()], [+(\texttt{Num}, \texttt{Bool})])$.

By splitting $\mathbb{J}$ in different ways, we may prove the following judgments (among others), all of them correct, each exhibiting a different trade-off of size and precision. For example, the first type is the only one that fully describes all the possible different shapes of the arrays in the data. The second type indicates the existence of some "pure integers" array, while the third one indicates the presence of empty arrays. The fourth one is the less informative but is still sound, since every array in the collection meets that description.

i) $\vdash \mathbb{J} :_u +\,(\texttt{Num},\ [+(\texttt{Num})],\ [+()],\ [+(\texttt{Num}, \texttt{Bool})]\,)$
ii) $\vdash \mathbb{J} :_u +(\,\texttt{Num},\ [+(\texttt{Num})],\ [+(\texttt{Num}, \texttt{Bool})]\,)$
iii) $\vdash \mathbb{J} :_u +(\,\texttt{Num},\ [+()],\ [+(\texttt{Num}, \texttt{Bool})]\,)$
iv) $\vdash \mathbb{J} :_u +(\,\texttt{Num},\ [+(\texttt{Num}, \texttt{Bool})]\,)$

4.2 The deterministic type system

The non-deterministic type system is a general framework that will be used to prove that every judgment that is generated by the process of interactive typing is sound. We formalize now a deterministic subsystem, that models how the user guides the behaviour of the system.

While the non-deterministic type system takes a non-deterministic choice of a split criterion for each instance of the (+) rule, our type checker splits the $\mathbb{J}$ collection according to a criterion that is defined by the user, and this is modeled by the deterministic system.

Our split criteria will be based on equivalence relations defined on JSON terms, as follows.

▶ **Definition 7** ($\mathcal{C}(E)$)**.** *For a given equivalence relation E defined on JSON terms, the split criterion $\mathcal{C}(E)$ maps a collection $\mathbb{J}$ to a partition of $\mathbb{J}$ according to E.*

We can now define two important equivalence relations which will be used to define two split criteria. They are *kind equivalence* $\mathcal{K}(J_1, J_2)$ and *label equivalence* $\mathcal{L}(J_1, J_2)$, and correspond to the equivalences $\mathcal{K}(\mathcal{S}_1, \mathcal{S}_2)$ and $\mathcal{L}(\mathcal{S}_1, \mathcal{S}_2)$ defined in [4] for structural types. Kind equivalence is the coarsest equivalence that can be used to define a split criterion, since it keeps all kind-homogeneous terms together. Label equivalence is the same as kind equivalence for base values and for arrays but, when records are compared, it only groups those records that have exactly the same keys, hence it is much finer. The formal definition of the two equivalences is obvious, and reported in the full version [3].

The simplest way to "determinize" the non-deterministic type system would be to choose a split criterion to be adopted in every instance of the (+) rule. We need, however, some more freedom than this, hence we adopt the notion of *determinizer*, which is a function from integer sequences to split criteria (Definition 8), that we use to associate a split criterion to each instance of the (+) rule in a proof.

Formally, we define the deterministic type system by adding a $\mathcal{D}$ parameter (a determinizer) to the rules, and by specializing the (+) rule to the following (+D) rule, that specifies that $\mathcal{D}(\epsilon)$, where ϵ is the empty sequence, is used to split $\mathbb{J}$, while the determinizers $next(\mathcal{D}, i)$, as defined in Definition 8, will be used in the subproofs.

$$\frac{(\mathbb{J}_1, \ldots, \mathbb{J}_n) = \mathcal{D}(\epsilon)(\mathbb{J}) \qquad \vdash^{next(\mathcal{D},i)} \mathbb{J}_i :_s \mathcal{S}_i \quad i = 1, \ldots, n}{\vdash^{\mathcal{D}} \mathbb{J} :_u +_{\mathcal{D}(\epsilon)}(\mathcal{S}_1, \ldots, \mathcal{S}_n)} \text{(+D)}$$

▶ **Definition 8** (Determinizer, $next(\mathcal{D}, i)$)**.** *A* determinizer $\mathcal{D}$ *is a function that maps any sequence, possibly empty, of positive integers $\omega = (i_1, \ldots, i_n)$ to a split criterion $\mathcal{D}(\omega)$. For any determinizer $\mathcal{D}$, $next(\mathcal{D}, i)$ is the determinizer that maps ω to $\mathcal{D}(i.\omega)$.*

A determinizer is hence a function that determines the split criterion used at each node of the type proof. By construction, every proof in the deterministic type system is also a proof in the non-deterministic one. In the full paper we show that the converse also holds: for every proof in the non-deterministic system a determinizer exists that generates that proof.

A determinizer may map any sequence to any split criterion, but, in practice, we will focus on very simple split criteria and very simple determinizers. Our system currently supports only three determinizers, L, K, LK, all of them employing equivalence-based criteria and a constant or almost-constant function.

▶ **Definition 9** (L, K, LK)**.** *L is the constant determinizer that maps every sequence ω to the split criterion $\mathcal{C}(\mathcal{L})$. K maps every sequence to $\mathcal{C}(\mathcal{K})$. LK maps the empty sequence to $\mathcal{C}(\mathcal{L})$ and any other sequence to $\mathcal{C}(\mathcal{K})$.*

Hereafter, we use the terms $\mathcal{L}$-type and $\mathcal{K}$-type, for a collection $\mathbb{J}$, to refer to the type that is inferred using, respectively, the L or the K determinizer. By Definition 9, L and K use, respectively, $\mathcal{C}(\mathcal{L})$ and $\mathcal{C}(\mathcal{K})$ as split criteria, everywhere in the proof. LK uses $\mathcal{C}(\mathcal{L})$ at the top level but $\mathcal{C}(\mathcal{K})$ everywhere else, that is, it expands like the $\mathcal{L}$-type at the top level, but computes a $\mathcal{K}$-type everywhere else.

4.3 The proof validity system

The aim of this paper is to provide a formal specification of a system that dynamically recomputes a type for a subset of the input data. We believe that the most direct approach is to describe it as a system that manipulates typing proofs and, to this aim, we need a notation to explicitly describe proofs and their manipulations. To this aim, we extend our judgments with a proof term that describes the proof itself, so that, rather than judgments $\vdash \mathbb{J} : \mathcal{T}$, that specify that the collection $\mathbb{J}$ has type $\mathcal{T}$, we will have ternary judgments with the form $\pi \vdash \mathbb{J} : \mathcal{T}$, that specify that π is a proof tree that proves that $\mathbb{J}$ has type $\mathcal{T}$.

More precisely, we first define a language of proof terms as follows.

$$\begin{array}{lcll} \pi^u & ::= & \langle +(\pi^s_1, \ldots, \pi^s_n)\rangle & n \geq 0 \\ \pi^s & ::= & \langle \mathbb{J}, \mathcal{B}\rangle \mid \langle \mathbb{J}, \{l_1 : \pi^u_1 \mathtt{q}_1, \ldots, l_n : \pi^u_n \mathtt{q}_n\}\rangle \mid \langle \mathbb{J}, [\pi^u]\rangle & \end{array}$$

Then, we define an inference system, based on two judgments that are mutually recursive $\pi^u \vdash \mathbb{J} :_u \mathcal{U}$ and $\pi^s \vdash \mathbb{J} :_s \mathcal{S}$ that specify when a proof term π is a proof (and not just a proof term), and, in that case, the fact that π proves that $\mathbb{J}$ has type $\mathcal{U}$. In the sequel we will use the metavariable π to range over both structural type proofs π^s and union type proofs π^u. The type inference rules are shown in Figure 2.

$$\text{(EMPTY)}\ \frac{}{\langle +(\,)\rangle \vdash \emptyset :_u +(\,)} \qquad \text{(+)}\ \frac{\mathcal{C} \text{ is a split criterion} \quad (\mathbb{J}_1, \ldots, \mathbb{J}_n) = \mathcal{C}(\mathbb{J}) \quad \pi_i \vdash \mathbb{J}_i :_s \mathcal{S}_i \quad i = 1, \ldots, n}{\langle +(\pi_1, \ldots, \pi_n)\rangle \vdash \mathbb{J} :_u +_{[\mathcal{C}]}(\mathcal{S}_1, \ldots, \mathcal{S}_n)} \qquad \text{(BASE)}\ \frac{kind(\mathbb{J}) = \mathcal{B}}{\langle \mathbb{J}, \mathcal{B}\rangle \vdash \mathbb{J} :_s \mathcal{B}}$$

$$\text{(REC)}\ \frac{kind(\mathbb{J}) = \{\,\} \quad \text{Let } (a_1, \ldots, a_n) = keys(\mathbb{J}) \quad \forall i = 1, \ldots, n.\ \pi_i \vdash \mathbb{J}/a_i :_u \mathcal{U}_i \quad \forall i = 1, \ldots, n.\ q_i = Q(\frac{|\mathbb{J}/a_i|}{|\mathbb{J}|})}{\langle \mathbb{J}, \{a_1 : \pi_1 q_1, \ldots, a_n : \pi_n q_n\}\rangle \vdash \mathbb{J} :_s \{a_1 : \mathcal{U}_1 q_1, \ldots, a_n : \mathcal{U}_n q_n\}} \qquad \text{(ARRAY)}\ \frac{kind(\mathbb{J}) = [\,] \quad \pi \vdash \mathbb{J}/[*] :_u \mathcal{U}}{\langle \mathbb{J}, [\pi]\rangle \vdash \mathbb{J} :_s [\mathcal{U}]}$$

Figure 2 Proof validity rules.

Given a proof π, in the full paper we define $\mathtt{JJ}(\pi)$ and $\mathtt{U}(\pi)$ as the unique collection $\mathtt{JJ}(\pi)$ and type $\mathtt{U}(\pi)$ such that $\pi \vdash \mathtt{JJ}(\pi) : \mathtt{U}(\pi)$. We then define a notion of path inside a proof or a type, where the steps (i), $[*]$ and a traverse, respectively, the union, array, and record constructors:

$$p \quad ::= \quad \epsilon \mid (i)/p \mid [*]/p \mid a/p$$

We then define a notion of subproof along a path $\pi \downarrow p$, and a notion of substitution $\pi[p \leftarrow \pi_1]$, that is only well defined when $\mathtt{JJ}(\pi_1) = \mathtt{JJ}(\pi \downarrow p)$, that is, a substitution will change the type but not the term. In the full paper we prove the fundamental property of this operation: if π and π_1 are valid proofs, then $\pi[p \leftarrow \pi_1]$ is a valid proof as well, where *valid* is defined by the rules of Figure 2. This property is the foundation of the formalization that we are going to present.

5 The Local-Retype Operation

We are now ready to define the local-retype operation. In our tool, the analyst has a collection $\mathbb{J}$ and a type $\mathcal{T}$ for $\mathbb{J}$, she/he clicks on a union type $\mathcal{U}$ inside $\mathcal{T}$ and selects the new determinizer $\mathcal{D}$ to use in order to re-type the subforest of $\mathbb{J}$ that corresponds to the type $\mathcal{U}$.

In general, this subforest does not correspond to a subsequence of the JSON collection: when the analyst selects the type of the *authors* field of a record type, the objects that correspond to the *authors* key are scattered through all those records that actually contain an *authors* field. However, if we consider a proof tree π such that $\pi \vdash \mathbb{J} : \mathcal{T}$, then these objects are all collected in the only premise of the (REC) rule that infers a type for $\mathbb{J}/authors$. Hence, we may formalize this action as follows: the analyst chooses a path p inside the type $\mathcal{T}$ and a new determinizer $\mathcal{D}$, and the system computes a new proof π' for the collection that corresponds to $\pi \downarrow p$, substitutes π' to $\pi \downarrow p$, and visualizes the type that corresponds to the new proof. The analysts operate on types, but the system is manipulating the corresponding proofs. We formalize this through the *retype* operation, as follows.

▶ **Definition 10** (*retype*$(\pi, p, \mathcal{D})$)**.** *For any proof π such that $\pi \vdash \mathbb{J} : \mathcal{T}$, for any p such that $\mathcal{T} \downarrow p$ is defined, for any determinizer $\mathcal{D}$, retype$(\pi, p, \mathcal{D})$ is defined as follows , where $\pi_{\mathcal{D}}$ is uniquely determined by $\mathcal{D}$, π and p :*

$$(\exists \mathcal{T}'.\ \pi_{\mathcal{D}} \vdash^{\mathcal{D}} \mathit{JJ}(\pi \downarrow p) : \mathcal{T}') \ \Rightarrow\ \mathit{retype}(\pi, p, \mathcal{D}) \triangleq \pi[p \leftarrow \pi_{\mathcal{D}}]$$

In other terms, the analyst indicates a path p, the system retrieves the corresponding collection $\mathtt{JJ}(\pi \downarrow p)$ inside the current proof π, computes a $\mathcal{D}$-proof $\pi_{\mathcal{D}}$ for that collection, and substitutes $\pi \downarrow p$ with $\pi^{\mathcal{D}}$. From the analyst's viewpoint, the type at $\mathcal{T} \downarrow p$ has been substituted with a $\mathcal{D}$-type for the corresponding collection.

In the full paper, we extend this definition to a *retype*$^*(\pi, \sigma)$ operation that executes a sequence $\sigma = (\,(p_1, \mathcal{D}_1), \ldots, (p_n, \mathcal{D}_n)\,)$ of retyping steps, and we prove that, for any such sequence, the resulting type is always a correct description of the input collection and a tight description, where every path in the type actually corresponds to some piece of data. Moreover, we prove that the split criteria that are present in the type actually correspond to how the collections have been split in the proof. In other terms, we prove that the specific sequence of steps is irrelevant, hence the final type that is displayed to the analyst only depends on the terms in the collection. For space reason, we only present here the most important results, but leave the discussion and the proofs to the full paper.

▶ **Theorem 11** (Type soundness of *retype*$^*()$)**.** *For any proof π such that $\pi \vdash \mathit{JJ}(\pi) : \mathtt{U}(\pi)$, for any sequence σ of retyping steps, the proof retype$^*(\pi, \sigma)$ is still a proof for $\mathit{JJ}(\pi)$:*

$$\mathit{retype}^*(\pi, \sigma) = \pi_n \ \Rightarrow\ \pi_n \vdash \mathit{JJ}(\pi) : \mathtt{U}(\pi_n)$$

▶ **Corollary 12.** *For any π such that $\pi \vdash \mathit{JJ}(\pi) : \mathtt{U}(\pi)$, for any sequence σ of retyping steps such that retype$^*(\pi, \sigma) = \pi_r$, for any path p:*

1. $[\![\, \mathit{JJ}(\pi_r \downarrow p) \,]\!] \subseteq [\![\, \mathtt{U}(\pi_r \downarrow p) \,]\!]$ — *Soundness of retype$^*()$*
2. *if $\pi \downarrow p$ is not $+()$:* $[\![\, \mathit{JJ}(\pi_r \downarrow p) \,]\!] \neq \emptyset$ — *Tightness of retype$^*()$*
3. *if $\pi_r \downarrow p = +_{\mathcal{C}}(\pi_1, \ldots, \pi_m)$:* $(\,\mathit{JJ}(\pi_1), \ldots, \mathit{JJ}(\pi_m)\,) = \mathcal{C}(\mathit{JJ}(\pi_r \downarrow p))$ — *Soundness of the split criterion*

6 Implementation

According to our definition, any time the analyst expands or contracts a node in the type, all data associated to that node are type-checked again. Since we are dealing with massive data collections, and we aim for a very fast response, this approach is not acceptable.

In our implementation, we do a different thing. We currently support the three determinizers K, L, and LK of Definition 9, and the L-type contains enough information to rebuild the K-type and the LK-type. Hence, we first compute and store the L-type $\mathcal{U}_L$ of the dataset $\mathbb{J}$, and we compute the K-type $\mathcal{U}_K$, which is the first one to be visualized, starting from $\mathcal{U}_L$. Then, rather than keeping track of a proof $\pi \vdash \mathbb{J} : \mathcal{U}_K$, we store a type-level abstract proof $\pi_{LK} \vdash \mathcal{U}_L : \mathcal{U}_K$, formalized in a type-to-type inference system specified below. When we need to recompute the type of a subcollection $\texttt{JJ}(\pi \downarrow p)$, rather than type-checking the subcollection, which may be extremely big, we compute its type starting from the collection of types $\texttt{U}(\pi_{LK} \downarrow p)$, which is a subforest of $\mathcal{U}_L$, and is an abstract description of $\texttt{JJ}(\pi \downarrow p)$.

In the full paper, in this section we present this approach, which is crucial for the applicability of our tool, and prove its correctness. We recap here the contents of the section.

We want to specify a set of conditions that make it possible to compute an abstract type from a detailed type without accessing the data. We focus our attention to a specific class of determinizers, those whose split condition is defined in terms of an equivalence, and, for the detailed type, we also ask that the split condition is constant, as it happens for the L determinizer, which constantly uses the $\mathcal{C}(\mathcal{L})$ split criterion. We then define a judgment $\vdash^{\mathcal{D}} \mathbb{J} :: \mathbb{S}$ that specifies that a collection of terms $\mathbb{J}$ is "described" by a collection of structural types $\mathbb{S}$. We can now define a notion of *refinement*, that specifies a condition that is sufficient in order to compute an abstract type starting from the detailed type.

The notion is based on the existence of a *Type-level split criterion for $\mathcal{D}$ and $\mathcal{C}$* $tsplit^{\mathcal{C}}_{\mathcal{D}}(\mathbb{S})$, that is, a function that splits the collection $\mathbb{S}$ describing $\mathbb{J}$ in the "same way" as $\mathcal{C}$ would have split $\mathbb{J}$. If, for a determinizer $\mathcal{E}$, for any ω, we have a type-level split criterion $tsplit^{\mathcal{E}(\omega)}_{\mathcal{D}}(\mathbb{S})$, then $\mathcal{D}$ *refines* $\mathcal{E}$, and we prove that this is sufficient to compute the $\mathcal{E}$-type from the $\mathcal{D}$-type.

To this aim, we present a set of rules to compute an $\mathcal{E}$-type from a $\mathcal{D}$-type if $\mathcal{D}$ refines $\mathcal{E}$, and the corresponding proof of correctness. This set of rules, reported in Figure 3, defines a *type-to-type* system, where the type collections at the left hand side substitute the terms that they represent, and the operators $\mathbb{S}/a$, $qual(\mathbb{S}, a)$, $keys(\mathbb{S})$, $\mathbb{S}/[*]$ represent the type-level lifting of the corresponding term operators.

$$\frac{}{\vdash^{\mathcal{E}}_{\mathcal{D}} +_C(\,) :_u +_{\mathcal{E}(\epsilon)}(\,)} \text{(EMPTY)} \qquad \frac{kind(\mathbb{S}) = \mathcal{B}}{\vdash^{\mathcal{E}}_{\mathcal{D}} \mathbb{S} :_s \mathcal{B}} \text{(BASE)} \qquad \frac{kind(\mathbb{S}) = [\,] \quad \vdash^{\mathcal{E}}_{\mathcal{D}} \mathbb{S}/[*] :_u \mathcal{U}}{\vdash^{\mathcal{E}}_{\mathcal{D}} \mathbb{S} :_s [\mathcal{U}]} \text{(ARRAY)}$$

$$\frac{(\mathbb{S}^1, \ldots, \mathbb{S}^n) = tsplit^{\mathcal{E}(\epsilon)}_{\mathcal{D}}(\mathbb{S}) \quad \vdash^{next(\mathcal{E},i)}_{\mathcal{D}} \mathbb{S}^i :_s \mathcal{S}'_i \quad i = 1, \ldots, n}{\vdash^{\mathcal{E}}_{\mathcal{D}} \mathbb{S} :_u +_{\mathcal{E}(\epsilon)}(\mathcal{S}'_1, \ldots, \mathcal{S}'_n)} \text{(+D)}$$

$$\frac{kind(\mathbb{S}) = \{\,\} \quad \text{Let } (a_1, \ldots, a_n) = keys(\mathbb{S}) \quad \forall i = 1, \ldots, n.\ \vdash^{\mathcal{E}}_{\mathcal{D}} \mathbb{S}/a_i :_u \mathcal{U}_i \quad q_i = qual(\mathbb{S}, a_i)}{\vdash^{\mathcal{E}}_{\mathcal{D}} \mathbb{S} :_s \{a_1 : \mathcal{U}_1 q_1, \ldots, a_n : \mathcal{U}_n q_n\}} \text{(REC)}$$

Figure 3 The t2t system: rules to infer an $\mathcal{E}$-type from a $\mathcal{D}$-type.

We then prove that the type-to-type system can be used to correctly compute the $\mathcal{E}$-type of any collection starting from its $\mathcal{D}$-type.

▶ **Property 13** (Soundness and completeness of $\vdash^{\mathcal{E}}_{\mathcal{D}} \mathcal{U}_D :_u \mathcal{U}_E$)**.** *For any $\mathbb{J}$, $\mathbb{S}$, $\mathcal{D}$, $\mathcal{E}$, $\mathbb{J}_c$, $\mathcal{U}_E$, $\mathcal{S}_E$, where $\mathbb{J}_c$ is kind-homogeneous, if $\mathcal{D}$ refines $\mathcal{E}$, then:*

$$\vdash^{\mathcal{D}} \mathbb{J} :: \mathbb{S} \quad \Rightarrow \quad (\ \vdash^{\mathcal{E}}_{\mathcal{D}} \mathbb{S} :_u \mathcal{U}_E \quad \Leftrightarrow \quad \vdash^{\mathcal{E}} \mathbb{J} :_u \mathcal{U}_E\ )$$
$$\vdash^{\mathcal{D}} \mathbb{J}_c :: \mathbb{S} \quad \Rightarrow \quad (\ \vdash^{\mathcal{E}}_{\mathcal{D}} \mathbb{S} :_s \mathcal{S}_E \quad \Leftrightarrow \quad \vdash^{\mathcal{E}} \mathbb{J}_c :_s \mathcal{S}_E\ )$$

7 Related Work

While the problem of inferring structural information from JSON collections has recently gained a momentum [8, 1, 2, 13, 9, 14], we are not aware of any approach that is interactive. In the XML realm, among the plethora of schema inference approaches [5, 10, 7, 11, 15, 12], only the one presented [15] is interactive since it relies on user intervention for recognizing regular expressions that are similar enough to be merged and for guiding the system to derive sophisticated schema expressing inheritance and restrictions, two constructs that are difficult to infer in a purely automatic fashion. While this approach bears some resemblance with our work in that both give the user some form of freedom to decide when two types are similar, the interaction presented in [15] is only meant to guide the system during the schema inference and does not allow for exploring different parts of an already existing schema with different precision lenses. Another notable difference with our approach is the lack of a mechanism for reasoning about type "similarity" like an equivalence class.

8 Conclusions

When semistructured JSON data are retrieved from the web, the presence of an automatic tool that can infer a schema is very useful, but the same data can be described with schemas with different size-precision trade offs, and no trade-off is optimal in general. Moreover, it is often the case that one is mostly interested in a specific part of the data, hence we designed and implemented a system where the analyst can interactively decide which parts of the data should be described with a greater or lower level of detail.

In this paper we have studied the formal foundations of this approach. The main novelty of this formal study is the use of specific precision criteria on specific parts of the data, that are identified by acting on a node of the syntax tree of a type. This aspect has been formalized through the use of some technical tools such as the use of selectors such as $\mathbb{J}/a$ and $\mathbb{J}/[*]$ inside the type proofs, and the notion of proof terms. With these tools we have proved the soundness of the local retype operation that we implemented.

The basic ideas that we presented here admit many extensions. First of all, while we designed the foundations of an interactive tool for an elementary type system, all the extensions of that elementary type system that we briefly presented in [4] – such as type constraints, enumeration types, variant types, tuple types, keyset equivalence – can be easily added. In the same way, the approach that we defined here can be easily applied to the counting type system that we defined in [2], hence combining the interactivity that we presented here with the quantitative information that was presented in that paper. We are currently exploring these possibilities.

Finally, while in [4] and [2] we studied the use of equivalence relations in order to split collection, here we allow any criterion to be used. This opens the way to some new possibilities, such as a top-n split, where one separates and groups the n shapes that are most important and collapses the others, or even forms of value-based grouping, where different records are grouped in a way that depends on the values of the fields and not just on the types.

References

1 Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schema Inference for Massive JSON Datasets. In *EDBT '17*, 2017.

2 Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Counting Types for Massive JSON Datasets. In *DBPL '17*, 2017.

3 Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. A Type System for Interactive JSON Schema Inference, April 2019. URL: `https://hal.archives-ouvertes.fr/hal-02112560`.

4 Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive JSON datasets. *The VLDB Journal*, pages 1–25, 2019.

5 Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of Concise DTDs from XML Data. In *VLDB '06*, pages 115–126, 2006. URL: `http://dl.acm.org/citation.cfm?id=1164139`.

6 Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. Technical report, Internet Engineering Task Force (IETF), December 20017. Standards Track.

7 Radu Ciucanu and Slawek 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. `arXiv:1307.6348`.

8 Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Typing Massive JSON Datasets. In *XLDI '12, Affiliated with ICFP*, 2012.

9 Michael DiScala and Daniel J. Abadi. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *SIGMOD '16*, pages 295–310. ACM, 2016. `doi:10.1145/2882903.2882924`.

10 Dominik D. Freydenberger and Timo Kötzing. Fast Learning of Restricted Regular Expressions and DTDs. *Theory Comput. Syst.*, 57(4):1114–1158, 2015.

11 Markus Lohrey, Sebastian Maneth, and Carl Philipp Reh. Compression of Unordered XML Trees. In *ICDT'07*, pages 18:1–18:17, 2017. `doi:10.4230/LIPIcs.ICDT.2017.18`.

12 Irena Mlỳnková and Martin Nečaskỳ. Heuristic methods for inference of XML schemas: Lessons learned and open issues. *Informatica*, 24(4):577–602, 2013.

13 Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON Schema. In *WWW '16*, pages 263–273, 2016. `doi:10.1145/2872427.2883029`.

14 Stefanie Scherzinger, Eduardo Cunha de Almeida, Thomas Cerqueus, Leandro Batista de Almeida, and Pedro Holanda. Finding and Fixing Type Mismatches in the Evolution of Object-NoSQL Mappings. In *Proceedings of the Workshops of the EDBT/ICDT 2016*, 2016. URL: `http://ceur-ws.org/Vol-1558/paper10.pdf`.

15 Julie Vyhnanovska and Irena Mlynkova. Interactive Inference of XML Schemas. In *Proceedings of the Fourth IEEE International Conference on Research Challenges in Information Science, RCIS 2010, Nice, France, May 19-21, 2010*, pages 191–202, 2010.