Typing Massive JSON Datasets

Dario Colazzo *
Université Paris Sud - INRIA
colazzo@lri.fr

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

Carlo Sartiani †
Università della Basilicata
sartiani@gmail.com

Abstract

Cloud-specific languages are usually untyped, and no guarantees about the correctness of complex jobs can be statically obtained. Datasets too are usually untyped and no schema information is needed for their manipulation.

In this paper we sketch a typing algorithm for JSON datasets. Our approach can be used to infer a *succinct* type from scratch for a collection of JSON objects, as well as to validate a dataset against a human-designed type and, if necessary, to adapt and improve this type.

1. Introduction

Cloud computing is a novel and very popular computing paradigm that aims at building extremely scalable and elastic applications working on huge datasets. This paradigm is based on the idea of using hundreds or thousands of low-end, unreliable, and cheap machines connected through standard network switches. The most popular incarnation of this paradigm is the Map/Reduce architecture [8], first introduced by Google and then adopted by companies like Facebook and Amazon.

While cloud computing applications can be written in standard programming languages like Java or C++, there has been much work in the development of special-purpose languages for the Cloud, such as Pig Latin [9] or Sawzall [10], that hide low-level details.

In cloud-specific languages types are usually optional, and programs may be deemed as correct at static time even if type errors are present: this may happens, for instance, when Pig Latin programs contain type casts. Hence, only weak guarantees about the correctness of complex jobs can be obtained at static time. This is particularly frustrating when multiple jobs form a single complex workflow processing huge datasets, as happens in clustering and machine learning computations. In this case, a statically undetected mismatch between the output of a job J_i and the input of the next job J_{i+1}, or between the output returned by Map worker and the expected input of reducers, can lead to incomplete and/or erroneous results, hence wasting large amounts of CPU time. These mismatches are hard to detect for the programmer, as she usually has to manually inspect the logs of Map and Reduce workers.

It is therefore crucial to extend cloud-specific languages with the typing mechanisms offered by modern programming languages. A preliminary step towards this direction is the design of a data typing algorithm that could automatically infer a type for a dataset or assist the programmer in designing it; key properties of the inferred types should be *succinctness* and *precision*, which are not easy to ensure in this setting.

* Dario Colazzo has been partially funded by the *Europa* 2012 EIT ICT Labs activity (RCLD12115-T1205A).

† Carlo Sartiani has been partially funded by RIL/2008.

Our Contribution In this paper we sketch a typing algorithm for JSON datasets [2]. The approach we propose here can be used to infer from scratch a *succinct* type for a collection of JSON objects, as well as to validate a dataset against a human-designed type and, if necessary, to adapt and improve this type.

The proposed algorithm consists of two stages.

1. In the first stage a Map/Reduce job processes the whole dataset and infers, for each JSON object, a type. In particular, during the Map phase, mappers examine all objects and, for each object, return a record containing an inferred type as the key and, as in WordCount, the value 1.

   Before the Reduce phase starts, records output by mappers are grouped by comparing their keys by means of a structural type equivalence algorithm.

   In the Reduce phase each reducer counts the number of elements for each equivalence class.

   The result of the first stage is then a set of pairs $< T_i; m_i >$, such that the type $\bigcup_i T_i$ precisely describes the input dataset, and each m_i counts the input values described by T_i.

2. The goal of the second stage is to *collapse* similar types by using a *type fusion* algorithm, without loosing too much precision. This second phase will hence produce a new collection of types U_j such that

   $$\begin{array}{ll}(i) & \bigcup_i T_i <: \bigcup_j U_j \text{ and} \\ (ii) & |\bigcup_j U_j| \leqslant |\bigcup_i T_i|,\end{array}$$

   so that $\bigcup_j U_j$ is a more general and succinct description than $\bigcup_i T_i$. [1] This phase is guided by an heuristic that depends on the numerosity m_i of each type.

2. Data Model and Type Language

JSON objects are unordered sets of name/value pairs, where names are unique strings and values comprise strings, booleans, chars, numbers, objects, and ordered lists of values (called *arrays*). Ordered lists are not necessarily homogeneous. JSON objects obey the following grammar, where n denotes a number, s a string, c a char, and l a string label.

$$\begin{array}{lll} o ::= & \{l : v, \ldots, l : v\} & \text{Objects} \\ v ::= & o & \\ & \mid\ [v, \ldots, v] & \text{Arrays} \\ & \mid\ v_s & \text{Simple values} \\ & \mid\ \epsilon & \text{Empty value} \end{array}$$

$$v_s ::= \mathbf{true} \mid \mathbf{false} \mid s \mid c \mid n$$

[1] $<:$ denotes the standard subtyping relation defined as $S <: T \iff [\![S]\!] \subseteq [\![T]\!]$.

We assume that records (i.e., objects) and lists (i.e., arrays) can be manipulated with traditional record and list operations: in particular, we assume that *record concatenation*, *field selection*, and *list concatenation* are available. These operations are defined as shown in Table 2.1, where $\uparrow$ denotes an error. Record and list concatenation can be lifted to sets in the usual way.

Our type language, vaguely inspired by that described by Benzaken et al. in [4], is shown below.

$$
\begin{array}{lll}
T ::= & \mathsf{B} & \\
\mid & \{l:T,\ldots,l:T\} & \text{Closed record type} \\
\mid & T \cdot T & \text{List concatenation} \\
\mid & T + T & \text{Union type} \\
\mid & T \circ T & \text{Record concatenation} \\
\mid & T* \mid T+ \mid T? \mid \epsilon &
\end{array}
$$

$$\mathsf{B} ::= \quad String \mid Bool \mid Char \mid Number \quad \text{Base types}$$

The semantics of types is standard: as usual, $[\![\_]\!]$ is the minimal function from types to sets of values that satisfies the following monotone equations (for the sake of simplicity, we omitted the semantics of base types), where π denotes a permutation of its domain and L_1, L_2 are metavariables for lists:

$$
\begin{array}{lcl}
[\![\epsilon]\!] & \triangleq & \{\epsilon\} \\
[\![\{l_1:T_1,\ldots,l_n:T_n\}]\!] & \triangleq & \{\{m_1:u_1,\ldots,m_n:u_n\} \mid \\
 & & \exists\pi:1..n \to 1..n.\forall i \in [1,n]: \\
 & & \pi(i)=h \implies l_i = m_h \wedge \\
 & & u_h \in [\![T_i]\!]\} \\
[\![T_1 \cdot T_2]\!] & \triangleq & [\![T_1]\!] \cdot [\![T_2]\!] \\
 & \triangleq & \{L_1 \cdot L_2 \mid L_1 \in [\![T_1]\!], L_2 \in [\![T_2]\!]\} \\
[\![T_1 + T_2]\!] & \triangleq & [\![T_1]\!] \cup [\![T_2]\!] \\
[\![T_1 \circ T_2]\!] & \triangleq & [\![T_1]\!] \circ [\![T_2]\!] \\
 & \triangleq & \{o_i \circ o_j \mid o_i \in [\![T_1]\!], o_j \in [\![T_2]\!]\} \\
[\![T*]\!] & \triangleq & [\![T]\!]* \\
[\![T+]\!] & \triangleq & [\![T]\!]+ \\
[\![T?]\!] & \triangleq & [\![T]\!]?
\end{array}
$$

Here, $T_1 \cdot T_2$ and $T_1 \circ T_2$ apply, respectively, list concatenation and record concatenation to all pairs of elements coming from T_1 and T_2. Type $T_1 + T_2$ is the union of T_1 and T_2. Type $T?$ takes the union of T with the empty-list type, while $T*$ and $T+$ are defined in the usual way, by iterating list concatenation and taking the union of the results of all the iterations. As in JSON objects, labels cannot be repeated in record types; hence, two record types T_1 and T_2 can be concatenated only if their label sets are disjoint.

3. Typing Algorithm

Our typing algorithm consists of two stages: during the first one, data are analyzed to infer a "raw" collection of types; during the second one, then, types are refined, through a process of *type fusion*, to decrease their size while preserving a good level of precision.

3.1 Type Inference

The first step of our algorithm is modelled as a Map/Reduce job analyzing all the objects in the dataset, according to the pseudocode shown in Figure 1. MAP processes each JSON object and infers a type for it by using the INFER procedure, based on the inference rules of Table 3.1. If an input type T is provided, then MAP matches each object against T and infers a new type only if the match fails.

Records returned by Map workers are grouped for the Reduce phase according to a structural type equivalence algorithm, described by the rules of Table 3.2. [2] These rules actually define a symbolic subtyping algorithm, and equivalence between T_1 and T_2 is verified by checking that $T_1 \lesssim T_2$ and $T_2 \lesssim T_1$. In the following we will use $T_1 \simeq T_2$ to indicate that T_1 and T_2 are structurally equivalent. [3]

Symbolic subtyping rules define a partial order relation among types; hence, if $T_1 \not\lesssim T_2$ and $T_2 \not\lesssim T_1$, we use the lexicographical order on the string representation of T_1 and T_2.

```
MAP(JSONObj o; Optional Type T)
1  if (T == NULL) or not ISMEMBER(o, T)
2      return < INFER(o); 1 >
3  else return < T; 1 >

REDUCE(< Type T; IntList list >
1  int card = 0
2  for each i ∈ list
3      card = card + 1
4  return < T; card >
```

Figure 1. Pseudocode of the Map/Reduce job.

Table 3.1. *Type inference rules.*

$$\frac{}{\vdash \mathbf{true} : Bool}\ (\text{TYPETRUEBOOL}) \qquad \frac{}{\vdash \mathbf{false} : Bool}\ (\text{TYPEFALSEBOOL})$$

$$\frac{}{\vdash n : Number}\ (\text{TYPENUMBER}) \qquad \frac{}{\vdash s : String}\ (\text{TYPESTRING})$$

$$\frac{}{\vdash c : Char}\ (\text{TYPECHAR}) \qquad \frac{\vdash v_i : T_i \quad i = 1,\ldots,n}{\vdash [v_1,\ldots,v_n] : T_1 \cdot \ldots \cdot T_n}\ (\text{TYPEARRAY})$$

$$\frac{\begin{array}{ll} \forall i = 1,\ldots,n: & \vdash l_i : String \\ \forall i,j = 1,\ldots,n: & i \neq j \implies l_i \neq l_j \\ \forall i = 1,\ldots,n: & \vdash v_i : T_i \end{array}}{\vdash \{l_1:v_1,\ldots,l_n:v_n\} : \{l_1:T_1,\ldots,l_n:T_n\}}\ (\text{TYPEREC})$$

EXAMPLE 3.1. *Consider the following four JSON objects:*

$\{$ id : 1,
age : 14,
$admin$: **false**,
$name$: *"John Smith"*,
$phone$: 31324378$\}$

$\{$ id : 2,
$name$: *"Edmond Dantes"*,
$email$: *"ed@mc.com"*,
$admin$: **true**$\}$

$\{$ id : 3,
$name$: *"Mattia Pascal"*,
$admin$: **false**,
age : 37,
$phone$: *"+333743227"*
$email$: *"mp@pir.net"*$\}$

$\{$ id : 4,
$name$: *"Amanda Clarke"*,
age : 26,
$admin$: **false**,
$phone$: 2123142222$\}$

During the Map phase, each object is inspected and a distinct type is created. In this case, the Map phase returns the following

[2] For the sake of simplicity, we omitted the rules for base types.

[3] We exploit here the ability of Hadoop [1] to accept any Java object implementing the Comparable Java interface as a key.

Table 2.1. *Record and list operations.*

$$\{l_1 : v_1, \ldots, l_n : v_n\}.l \triangleq \begin{cases} v_i & \text{if } \exists i \in [1,n] : l = l_i \\ \uparrow & \text{otherwise} \end{cases}$$

$$\{l_1 : v_1, \ldots, l_n : v_n\} \circ \{m_1 : u_1, \ldots, m_p : u_p\} \triangleq \begin{cases} \{l_1 : v_1, \ldots, l_n : v_n, m_1 : u_1, \ldots, m_p : u_p\} & \text{if } \{l_1, \ldots, l_n\} \cap \{m_1, \ldots, m_p\} = \emptyset \\ \uparrow & \text{otherwise} \end{cases}$$

$$[v_1, \ldots, v_n] \cdot [u_1, \ldots, u_p] \triangleq [v_1, \ldots, v_n, u_1, \ldots, u_p]$$

Table 3.2. *Symbolic subtyping.*

ϵ	$\lesssim$	ϵ
ϵ	$\gtrsim$	$T*$
$\{l_1 : T_1, \ldots, l_n : T_n\}$	$\gtrsim$	$\{m_1 : U_1, \ldots, m_n : U_n\}$ if $\exists \pi : 1..n \to 1..n . \forall i \in [1,n] : l_i = m_{\pi(i)} \wedge T_i \lesssim U_{\pi(i)}$
$T_1 \cdot T_2$	$\lesssim$	$U_1 \cdot U_2$ if $T_1 \lesssim U_1$ and $T_2 \lesssim U_2$
T_1	$\lesssim$	$U_2 + U_3$ if $T_1 \lesssim U_2$ or $T_1 \lesssim U_3$ with $T_1 \neq V_1 + V_2$
$T_1 + T_2$	$\lesssim$	U if $T_1 \lesssim U$ and $T_2 \lesssim U$
T	$\gtrsim$	$U*$ if $T \lesssim U$
$T*$	$\gtrsim$	$U*$ if $T \lesssim U*$
$T_1 \cdot T_2$	$\gtrsim$	$U*$ if $T_1 \lesssim U*$ and $T_2 \lesssim U*$
$T_1 \circ T_2$	$\lesssim$	$U_1 \circ U_2$ if $T_1 \lesssim U_1$ and $T_2 \lesssim U_2$

four types:

$$T_1 = \{\ id : Number,\ age : Number,\ admin : Bool,\ name : String,\ phone : Number\}$$
$$T_2 = \{\ id : Number,\ name : String,\ email : String,\ admin : Bool\}$$
$$T_3 = \{\ id : Number,\ name : String,\ admin : Bool,\ age : Number,\ phone : String,\ email : String\}$$
$$T_4 = \{\ id : Number,\ name : String,\ age : Number,\ admin : Bool,\ phone : Number\}$$

Each type is adorned with the number 1.

Before the Reduce phase is activated, types are compared by using the structural equivalence algorithm, which discovers that the first and the fourth type are equivalent. T_1 and T_4, are, then, collapsed; as a consequence, the Reduce phase takes as input the pairs $< T_1; \{1,1\} >$, $< T_2; \{1\} >$, and $< T_3; \{1\} >$, updates the cardinality information in each pair, and returns the pairs $< T_1; 2 >$, $< T_2; 1 >$, and $< T_3; 1 >$.

3.2 Type Fusion

The second stage of our algorithm takes the collection of pairs $< type; card >$ returned by the first stage as input. The types in these pairs precisely describe the input dataset. These types, however, tend to be quite large and redundant, which makes them hard to use for an efficient typechecking. As they bear many structural similarities, it is however possible to derive a more succinct type, at the price of a loss of precision.

We envision here a type fusion process that starts by sorting $< type; card >$ pairs by ascending cardinality. The algorithm, then, selects the least representative type T_s (i.e., the type with least cardinality), and inspects the remaining types, in ascending cardinality order, to find types that can be successfully *fused* with T_s. When a candidate type T_x has been found, T_s and T_x are fused and the process is repeated until a size threshold is satisfied or no more fusions are possible.

Type fusion is guided by a set of fusion rules of the form $T_1 \mid T_2 \to U$, where $T_1 + T_2 <: U$. These rules exploit the structural similarities between types and cut redundant type fragments, as shown in following example.

EXAMPLE 3.2. *Consider the four JSON objects of Example 3.1. As already seen, the Map/Reduce job returns three pairs: $< T_1; 2 >$, $< T_2; 1 >$, and $< T_3; 1 >$. T_3, hence, is deemed as the less representative type in the collection and is compared with T_2 and T_1.*

During the comparison with T_2, the algorithm discovers that T_2 contains a strict subset of the fields of T_3. The algorithm, hence, applies the rule $U_1 \circ U_2 \mid U_1 \to U_1 \circ U_2?$, where $U_1 = T_2$ and $U_2 = \{age : Number, phone : String\}$.

The resulting type $U_1 \circ U_2?$, having now cardinality 2, is then matched against T_1, which has no email field and a different type for the phone field. The algorithm first decomposes[4] $U_1 \circ U_2?$ as $V_1 \circ V_2 \circ V_3 \circ V_4$, where:

$$V_1 = \{id : Number, name : String, admin : Bool\}$$
$$V_2 = \{email : String\}$$
$$V_3 = \{phone : String\}?$$
$$V_4 = \{age : Number\}?$$

The algorithm, then, modifies the type of the phone field and makes V_2 optional, hence returning the following type:

$$\{id : Number, name : String, admin : Bool\} \circ \{email : String\}? \circ \{phone : String + Number\}? \circ \{age : Number\}?$$

4. Discussion and Conclusions

The typing algorithm we sketched in the previous sections is in its very infancy, and there are many open issues that must be addressed and discussed.

1. Our type language is based on closed record types; an alternative solution would be the use of *open record types* that define only the minimal set of fields that an object must contain. Open record types have been extensively studied in the past (see [6]), and their properties are well-known. To some extent these types could be used to derive more succinct types, but we must better explore the trade-off between succinctness and precision.
2. Our type fusion algorithm is based on a greedy strategy in which types are sorted according to their actual cardinality, in order to maximize precision for the most representative types. Correctness of this algorithm must be formally proved, and the degree of succinctness and precision of returned types must be measured by means of experiments. An alternative approach

[4] This decomposition returns a supertype of the original type.

would be the use of a clustering algorithm that clusters types on the basis of a similarity metrics.

3. The third issue is tightly related to the previous one and concerns the notion of type similarity. Our idea is to use a metrics based on the structure of types; however, we want to explore also more *semantic* approaches based on constraints [7] or automata. All these approaches must be investigated not only from a theoretical point of view, but also from an experimental perspective.

4. The type fusion rules we envisioned here are rewriting rules that are based, again, on the structure of types. Ideally, these rules should be computationally not expensive, while forming a set that is rich enough to capture the most interesting and frequent cases.

   As for the type similarity measure, we also have to explore alternative type fusion approaches based on constraints or automata.

5. The last issue concerns the structure of the second stage of the algorithm. We sketched this stage as a sequential activity where all types are processed on a single machine. It could be very interesting to explore the possibility to parallelize this stage too and to implement it on cloud computing architectures [3, 8].

As a final remark, we observe that JSON bears many similarities with XML. In the recent past many approaches for deriving a DTD from a collection of XML documents have been proposed (see [5], for instance). These approaches usually rely on automata and are computationally expensive. It could be very interesting to understand if these approaches can be adapted to JSON and if they can be parallelized and/or improved by leveraging on the specific properties of JSON.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments.

References

[1] Hadoop. `http://hadoop.apache.org/`.

[2] Javascript object notation (JSON). `http://json.org`.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/Pacts: a programming model and execution framework for web-scale analytical processing. In J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors, *SoCC*, pages 119–130. ACM, 2010. ISBN 978-1-4503-0036-0.

[4] V. Benzaken, G. Castagna, K. K. Nguyen, and J. Siméon. The next 700 NoSQL languages. Manuscript Draft, nov 2011.

[5] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, 35(2), 2010.

[6] L. Cardelli. Extensible records in a pure calculus of subtyping. *Theoretical Aspects of Object-Oriented Programming*, pages 373–425, 1994.

[7] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. *Inf. Syst.*, 34(7):643–656, 2009.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 1099–1110. ACM, 2008. ISBN 978-1-60558-102-6.

[10] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4): 277–298, 2005.