Typing Massive JSON Datasets

Dario Colazzo
Université Paris Sud - INRIA

Giorgio Ghelli
Università di Pisa

Carlo Sartiani
Università della Basilicata

Outline

- Introduction & Motivation
- Data model & Type language
- Typing approach
- Conclusions and future work

INTRODUCTION & MOTIVATION

Cloud Computing

- Cloud computing is a very popular computing paradigm
  - Clusters of low-end, unreliable, cheap machines
- Applications
  - Data Storage
    - DropBox, SkyDrive, Google Drive, iCloud
  - Data Analysis
    - Facebook, Yahoo!, Google

Programming Data Intensive Applications

- Traditional programming languages (e.g., Java, C++, C#, etc.)
  - The programmers must deal with the details of the specific cloud architecture
    - map, reduce, local sort, combiner, etc. in Hadoop/Java
- Languages for the cloud
  - A mix between scripting languages and declarative database programming languages
  - They hide the deatails of the underlying architecture (e.g., Sawzall, Pig Latin)

Cloud Languages and Static Analysis

- Support for static analysis and typechecking is usually limited
  - Types for input data are optional
  - Many controls are deferred at runtime
  - There is no validation of the input data against a schema
- Consequences
  - Jobs can raise dynamic type errors

Example: Pig Latin

- Pig Latin types
  - Base types
    - int, long, double, bytearray, chararray
  - Map types
    - Used for describing records
  - Tuple types
    - Record types without types for the fields (only field labels are specified)
  - Bag types

Example: Pig Latin

- Input.txt: `(yahoo,25) (facebook,15) (twitter,7)`
- A program with a wrong schema:

```
data = LOAD 'input.txt' AS (query:INT,count:CHARARRAY);
data2 = FOREACH data GENERATE TOKENIZE(data.count);
STORE data2 INTO 'EXAMPLE1';
```

- This program is deemed as type-correct
- A run-time error is raised, as `TOKENIZE` only accepts strings as its input

Objectives

- To automatically derive succinct and precise schema information from large JSON datasets
  - Succinctness has a direct impact on efficiency and effectiveness of typechecking
  - Avoiding type errors related to imprecise types
  - A concise but precise supertype
- To extend and improve existing type systems of cloud languages

DATA MODEL AND TYPE LANGUAGE

Data Model

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

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

Type Language

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

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

Type Semantics

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

Type Equivalence

- We need a type equivalence notion for later use
  - Polynomial time
- We base our type equivalence on a set of syntactical subtyping rules
  - Correct but not complete
  - Polynomial time by using dynamic programming
- When two types are not comparable, we resort to a lexicographical comparison between types

Subtyping Rules

$$
\begin{array}{lcll}
\epsilon & \lesssim & \epsilon & \\
\epsilon & \lesssim & T* & \\
\{l_1 : T_1, \ldots, l_n : T_n\} & \lesssim & \{m_1 : U_1, \ldots, m_n : U_n\} & \text{if } \exists \pi : 1..n \to 1..n. \forall i \in [1,n] : \\
 & & & l_i = m_{\pi(i)} \wedge T_i \lesssim U_{\pi(i)} \\
T_1 \cdot T_2 & \lesssim & U_1 \cdot U_2 & \text{if } T_1 \lesssim U_1 \text{ and } T_2 \lesssim U_2 \\
T_1 & \lesssim & U_2 + U_3 & \text{if } T_1 \lesssim U_2 \text{ or } T_1 \lesssim U_3 \text{with } T_1 \neq V_1 + V_2 \\
T_1 + T_2 & \lesssim & U & \text{if } T_1 \lesssim U \text{ and } T_2 \lesssim U \\
T & \lesssim & U* & \text{if } T \lesssim U \\
T* & \lesssim & U* & \text{if } T \lesssim U* \\
T_1 \cdot T_2 & \lesssim & U* & \text{if } T_1 \lesssim U* \text{ and } T_2 \lesssim U* \\
T_1 \circ T_2 & \lesssim & U_1 \circ U_2 & \text{if } \exists \pi : 1..2 \to 1..2. \forall i \in [1,2] : T_i \lesssim U_{\pi(i)}
\end{array}
$$

TYPING APPROACH

Typing Algorithm

- Our typing algorithm comprises two stages
- First stage
  - We analyze each JSON object and infer a precise type for it
- Second stage
  - We fuse the collection of types obtained from the first stage so to get a more succinct type
  - Fusion is governed by a set of fusion rules

First Stage

- A Map/Reduce job
- In the Map phase we infer a type for each JSON object
  - We also record cardinality information (as in WordCount)
- In the Reduce phase equivalent types are grouped together and cardinality is updated
- The output of this job is a collection of pairs $<T_i; card_i>$
  - T_i is a type
  - $card_i$ is the number of object in the dataset having type T_i
  - $\bigcup_i T_i$ is the type for the objects in the dataset

Map/Reduce Job

$\textsc{Map}(JSONObj\ \ o;\ Optional\,Type\ \ T)$

if $(T == \textsc{null})$ **or not** $\textsc{isMember}(o, T)$
    return $< \textsc{Infer}(o); 1 >$
else return $< T; 1 >$

$\textsc{Reduce}(< Type\ \ T; IntList\ \ list >$

int $card\ =\ 0$
for each $i \in list$
    $card\ =\ card + 1$
return $< T; card >$

Typing Inference Rules

(TypeTrueBool)

$$\frac{}{\vdash \ \mathbf{true} \ : \ Bool}$$

(TypeFalseBool)

$$\frac{}{\vdash \ \mathbf{false} \ : \ Bool}$$

(TypeNumber)

$$\frac{}{\vdash \ n \ : \ Number}$$

(TypeString)

$$\frac{}{\vdash \ s \ : \ String}$$

(TypeChar)

$$\frac{}{\vdash \ c \ : \ Char}$$

(TypeArray)

$$\frac{\vdash \ v_i \ : \ T_i \quad i = 1, \ldots, n}{\vdash \ [v_1, \ldots, v_n] \ : \ T_1 \cdot \ldots \cdot T_n}$$

(TypeRec)

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

First Stage Example

- 4 JSON objects

$\{\ \ id : 1,$
$age : 14,$
$admin : \textbf{false},$
$name : \text{“John Smith”},$
$phone : 31324378\}$

$\{\ \ id : 2,$
$name : \text{“Edmond Dantes”},$
$email : \text{“ed@mc.com”},$
$admin : \textbf{true}\}$

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

$\{\ \ id : 4,$
$name : \text{“Amanda Clarke”},$
$age : 26,$
$admin : \textbf{false},$
$phone : 2123142222\}$

First Stage Example

- Map phase inferred types

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

- Reduce phase output: <T_1;{2}>, <T_2; {1}>, <T_3; {1}>

Second Stage

- Types obtained from the first stage are fused together
  - More succinct types
  - Loss of precision
- Fusion is performed according to fusion rules
- A fusion rule $\langle T_1 \mid T_2 \rangle \rightarrow T_3$ is a rewriting rule such that
  - $T_1 + T_2 \lesssim T_3$
  - $| T_1 + T_2 | \geq |T_3|$

Fusion Rules

- Fusion rules should be easy to check and to evaluate
- We have a provisional set of rules
  - Regular expression rules
    - Simplification rules
    - Subtyping rules
    - General rules
  - Record type rules

Regular Expression Rules

	Simplification rules	
1)	$T \mid \epsilon \to T$	if T is nullable
2)	$T \mid \epsilon \to T?$	
3)	$T+ \mid \epsilon \to T*$	
4)	$T \cdot \epsilon \mid U \to T + U$	
5)	$\epsilon \cdot T \mid U \to T + U$	
6)	$T \mid T \to T$	
	Subtyping rules	
7)	$T \mid U \to U$	if $T \sim U$ and $\lvert T\rvert \geqslant \lvert U\rvert$
8)	$T \mid U \to U$	if $T \lesssim U$ and $\lvert T\rvert \geqslant \lvert U\rvert$
	General rules	
9)	$T \mid T \cdot U \to T \cdot U?$	
10)	$T \cdot U \mid T \cdot V \to T \cdot (U + V)$	
11)	$T \mid U \cdot T \to U? \cdot T$	
12)	$U \cdot T \mid V \cdot T \to (U + V) \cdot T$	

Record Type Rules

$$
\begin{array}{ll}
13) & \{l_1 : T_1\} \circ U \mid \{l_1 : T_2\} \circ V \to \{l_1 : T_1 + T_2\} \circ (U + V) \\
14) & T \mid T \circ U \to T \circ U? \\
15) & T \circ U \mid T \circ V \to T \circ (U + V)
\end{array}
$$

Type Fusion Algorithm

- An heuristic algorithm that focuses on types with low cardinality
  - Types with cardinality greater than a given threshold are ignored to improve the overall precision
- Types are ordered by ascending cardinality into a priority queue
- At each iteration
  - The type with the lowest cardinality is popped
  - The type is compared with the other types in the queue to see if a fusion rule is applicable
- The algorithm halts when no more fusions are possible or a given threshold is satisfied

Example

- Output of the reduce phase: <T$_1$;{2}>, <T$_2$; {1}>, <T$_3$; {1}>
- Record types are splitted into concatenations of single-field record types

$$
\begin{aligned}
T_1 = \quad & \{ \quad id : Number\}\circ \\
& \{ \quad age : Number\}\circ \\
& \{ \quad admin : Bool\}\circ \\
& \{ \quad name : String\}\circ \\
& \{ \quad phone : Number\} \\
T_3 = \quad & \{ \quad id : Number\}\circ \\
& \{ \quad name : String\}\circ \\
& \{ \quad admin : Bool\}\circ \\
& \{ \quad age : Number\}\circ \\
& \{ \quad phone : String\}\circ \\
& \{ \quad email : String\}
\end{aligned}
$$

$$
\begin{aligned}
T_2 = \quad & \{ \quad id : Number\}\circ \\
& \{ \quad name : String\}\circ \\
& \{ \quad email : String\}\circ \\
& \{ \quad admin : Bool\}
\end{aligned}
$$

Example

- T_2 is selected and fused with T_3 by using Rule (14)

- Fused type V_1

$$
\begin{aligned}
V_1 = \quad & \{id : Number\} \circ \\
& \{name : String\} \circ \\
& \{email : String\} \circ \\
& \{admin : Bool\} \circ \\
& (\{age : number\} \circ \{phone : String\})?
\end{aligned}
$$

Example

- T_1 is selected and fused with V_1 by using Rule (15)
- Fused type V_2

$$
\begin{aligned}
V_2 = \quad & \{id : Number\}\circ \\
& \{name : String\}\circ \\
& \{admin : Bool\}\circ \\
& ((\{age : Number\} \circ \{phone : String\})+ \\
& (\{email : String\} \circ (\{age : Number\} \circ \{phone : Number\})?))
\end{aligned}
$$

Example

- V_2 is further simplified by applying fusion rules to its union types
- Rule (15) is applied to factorize {*age:Number*}
- Rule (13) is then applied to *phone* fields
- Output type

$$\begin{aligned} V_3 = \quad & \{id : Number\} \circ \\ & \{name : String\} \circ \\ & \{admin : Bool\} \circ \\ & (\{age : Number\} \circ (\{phone : String + Number\} \circ \{email : String\})?)? \end{aligned}$$

CONCLUSIONS AND FUTURE WORK

Research Status

- A very preliminary work
- We are working on
  - Defining a richer set of fusion rules
  - Improving the fusion algorithm
  - Designing a Map/Reduce fusion algorithm
  - Adapting our algorithm to the type languages of existing cloud languages

Conclusions

- An approach for typing massive datasets of JSON objects
  - Based on fusion rules
  - Adaptable to Map/Reduce