Static and Dynamic Semantics of NoSQL Languages

POPL 2013, Rome, Jan. 23-25 2013

Véronique Benzaken[1] Giuseppe Castagna[2]
Kim Nguyễn[1] Jérôme Siméon[3]

1 LRI, Université Paris-Sud, Orsay, France
2 CNRS, PPS, Univ Paris Diderot, Paris, France
3 IBM Watson Research, Hawthorne, NY, USA

Not Only SQL ?

SQL (and the Relational DBMS) are not good for everything

NoSQL is class of Database Management Systems that :

- Optimized for *scalability* and *performances*
- Often implemented on top of MapReduce* frameworks
  $\star$: distributed computations as the combination of node-local operations (Map) and global agregation of intermediary results (Reduce)
- Data-intensive applications

Not Only SQL ?

SQL (and the Relational DBMS) are not good for everything

NoSQL is class of Database Management Systems that :

- Optimized for *scalability* and *performances*
- Often implemented on top of MapReduce* frameworks
  * : distributed computations as the combination of node-local operations (Map) and global agregation of intermediary results (Reduce)
- Data-intensive applications

Writing applications directly with MapReduce is tedious

NoSQL programming languages

- High-level sequence operations (compiled to MapReduce)
- Often less expressive than SQL (no join for instance)
- Collection of tuples, key-value pairs (records), ...
- Flat or nested model

NoSQL programming languages

- High-level sequence operations (compiled to MapReduce)
- Often less expressive than SQL (no join for instance)
- Collection of tuples, key-value pairs (records), ...
- Flat or nested model

Problems :

- Not standard (yet) : Jaql, Pig/Latin, Sawzall, Unql, ...
- No formal semantics ⇒ hard to reason about the code
- Weak notion of schema (data types) ⇒ hard to specify program input/output (unusual for the DB community)
- No static typing (usual for the DB community)

NoSQL programming languages

- High-level sequence operations (compiled to MapReduce)
- Often less expressive than SQL (no join for instance)
- Collection of tuples, key-value pairs (records), ...
- Flat or nested model

Problems :

- Not standard (yet) : Jaql, Pig/Latin, Sawzall, Unql, ...
- No formal semantics ⇒ hard to reason about the code
- Weak notion of schema (data types) ⇒ hard to specify program input/output (unusual for the DB community)
- No static typing (usual for the DB community)

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
```

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
              transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
              transform x with { x.*, kind:"employee" }
)
```

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
              transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
              transform x with { x.*, kind:"employee" }
)
```

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
              transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
              transform x with { x.*, kind:"employee" }
)
```

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
              transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
              transform x with { x.*, kind:"employee" }
)
```

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
              transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
              transform x with { x.*, kind:"employee" }
)
```

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
              transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
              transform x with { x.*, kind:"employee" }
)
```

Question : what's the type of `union` ?

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
            transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
            transform x with { x.*, kind:"employee" }
)
```

Question : what's the type of `union` ?

$\forall\alpha\,.\,[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$ seems too restrictive…

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
              transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
              transform x with { x.*, kind:"employee" }
)
```

Question : what's the type of `union` ?

$\forall\alpha.[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$ seems too restrictive…

[any] $\rightarrow$ [any] $\rightarrow$ [any] seems too imprecise…

Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

```
//sequence of department records
depts = [ {depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [ {name:"Kim", depid:"210", salary:1000}, ... ]
union(
  depts -> filter each x (x.size > 50) ->
              transform x with { x.*, kind:"department" },
  empls -> filter each x (x.salary > 2000) ->
              transform x with { x.*, kind:"employee" }
)
```

Question : what's the type of `union` ?

$\forall\alpha.[\alpha] \to [\alpha] \to [\alpha]$ seems too restrictive...

$[\mathtt{any}] \to [\mathtt{any}] \to [\mathtt{any}]$ seems too imprecise...

"Some sophisticated dependent type" what about type inference ?

Outline

Using semantic subtyping to define JSON schema

A way to precisely describe the data

Filters

Recursive combinators that implement sequence iterators

Filters (Types)

Evaluating the program over an input type to compute the output type

Disclaimer : I'm "mostly" telling the truth (details in the paper)

JSON schema using regular expression types

What type can we give to `depts`, `employee` and the result of `union`?

```
type Depts = [{size:int, name:string, depnum:int}* ]
type Emp = [{name:string, depid: string, salary:int }* ]
```

JSON schema using regular expression types

What type can we give to `depts`, `employee` and the result of `union` ?

```
type Depts = [{size:int, name:string, depnum:int}* ]
type Emp = [{name:string, depid: string, salary:int }* ]
```

```
[ ({size:int, name:string, depnum:int; kind:"department" }
 |{name:string, depid:string, salary:int; kind:"employee"})* ]
```

JSON schema using regular expression types

What type can we give to `depts`, `employee` and the result of `union` ?

```
type Depts = [{size:int, name:string, depnum:int}* ]
type Emp = [{name:string, depid: string, salary:int }* ]
```

```
[ ({size:int, name:string, depnum:int; kind:"department" }
 |{name:string, depid:string, salary:int; kind:"employee"})* ]
```

How can we achieve that ?

Semantic subtyping 1/2

Definition (Types)

$$
\begin{array}{rlr}
t ::= & \texttt{int} \mid \texttt{string} \mid \ldots & \text{(basic types)} \\
\mid & \texttt{`nil} \mid 42 \mid \ldots & \text{(singleton types)} \\
\mid & (t,t) & \text{(products)} \\
\mid & \{\ell{:}t,\ldots,\ell{:}t\} & \text{(closed records)} \\
\mid & \{\ell{:}t,\ldots,\ell{:}t\ ,..\} & \text{(open records)} \\
\mid & t|t & \text{(union types)} \\
\mid & t\&t & \text{(intersection types)} \\
\mid & \neg t & \text{(negation type)} \\
\mid & \texttt{empty} & \text{(empty type)} \\
\mid & \texttt{any} & \text{(any type)} \\
\mid & \mu T.t & \text{(recursive types)} \\
\mid & T & \text{(recursion variable)}
\end{array}
$$

$$
\begin{array}{l}
\mu T.(\\
\quad \texttt{`nil} \\
\quad |(\{\texttt{"name"} : \texttt{string}\ ,..\}, T) \\
) \\
\equiv
\end{array}
$$

```
[ {name: string, ..}* ]
```

Semantic subtyping 2/2

Definition (Semantic subtyping)

$$s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!]$$

$[\![\_]\!]$: set-theoretic interpretation : a type is the set of the value that have that type

- Arbitrary regular expressions : `[char+ (int|bool)?]`
- **Semantic** equivalence of types :
  - `(int,int)|(2,4)` $\equiv$ `(int,int)`
  - `{"id":int,..}&{"here":bool,..}` $\equiv$ `{"id":int,"here":bool,..}`
  - `{"id":int,..}|{"id":bool,..}` $\equiv$ `{"id":(int | bool),..}`
- Decidable emptiness (since $s \leq t \Leftrightarrow s\&\neg t =$ `empty`)
- Decidable finiteness (since types are regular)

Basic expressions

Definition (Basic expressions)

$$
\begin{array}{rlr}
e ::= & c & \text{(constants)} \\
\mid & x & \text{(variables)} \\
\mid & (e, e) & \text{(pairs)} \\
\mid & \{e\!:\!e, \ldots, e\!:\!e\} & \text{(records)} \\
\mid & e + e & \text{(record concatenation)} \\
\mid & e \setminus \ell & \text{(field deletion)} \\
\mid & \mathrm{op}(e, \ldots, e) & \text{(built-in operators)} \\
\mid & f\ e & \text{(filter application)}
\end{array}
$$

Example

$\{\texttt{"age"}\!:\!30, \texttt{"name"}\!:\!\texttt{"Kim"}\} + \{\texttt{"age"}\!:\!31\} \rightsquigarrow \{\texttt{"age"}\!:\!31, \texttt{"name"}\!:\!\texttt{"Kim"}\}$

$\{\texttt{"age"}\!:\!30, \texttt{"name"}\!:\!\texttt{"Kim"}\} \setminus \texttt{"name"} \rightsquigarrow \{\texttt{"age"}\!:\!30\}$

$\{\texttt{strconcat("a", "ge")}\!:\!30\} \rightsquigarrow \{\texttt{"age"}\!:\!30\}$

Basic expression typing

$$[\textsc{Vars}]\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$[\textsc{Constant}]\quad \frac{}{\Gamma \vdash c : c}$$

$$[\textsc{Prod}]\quad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1, t_2)}$$

$$[\textsc{Foreign}]\quad \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \mathrm{op}(e_1, \ldots, e_n) : \mathtt{type}((\Gamma, x_1{:}t_1, \ldots, x_n{:}t_n), \mathrm{op}(x_1, \ldots, x_n))}$$

$$[\textsc{Rcd-Fin}]\quad \frac{\Gamma \vdash e : \ell_1 | \cdots | \ell_n \quad \Gamma \vdash e' : t}{\Gamma \vdash \{e{:}e'\} : \{\ell_1{:}t\} | \cdots | \{\ell_n{:}t\}}$$

$$[\textsc{Rcd-Inf}]\quad \frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t'}{\Gamma \vdash \{e{:}e'\} : \{..\}} \quad \begin{array}{c} t \leq \mathtt{string} \\ t \text{ is infinite} \end{array}$$

• • •

Filters

Definition (Filters)

$$
\begin{array}{rll}
f ::= & e & \text{(expression)} \\
\mid & p\texttt{=>}f & \text{(pattern)} \\
\mid & f \;\texttt{|}\; f & \text{(union)} \\
\mid & f\,\texttt{;}\,f & \text{(composition)} \\
\mid & \texttt{(}f\,\texttt{,}\,f\texttt{)} & \text{(product)} \\
\mid & \texttt{\{}\ell\texttt{:}f\texttt{,}\ldots\texttt{,}\ell\texttt{:}f\;\texttt{,}\;\texttt{..}\texttt{\}} & \text{(record)} \\
\mid & \texttt{let}\;\; X \;\texttt{=}\; f & \text{(rec. definition)} \\
\mid & X\;a & \text{(guarded rec.)}
\end{array}
$$

$$
\begin{array}{rll}
a ::= & x & \text{(variables)} \\
\mid & c & \text{(constants)} \\
\mid & \texttt{(}a\,\texttt{,}\,a\texttt{)} & \text{(pairs)} \\
\mid & \texttt{\{}\ell\texttt{:}a, \ldots, \ell\texttt{:}a\texttt{\}} & \text{(record)}
\end{array}
$$

$p ::=$ types with capture variables

Definition ((Big-step) semantics of filters)

$$\delta; \gamma \vdash_{eval} f(v) \rightsquigarrow r$$

δ : recursion variable environment
γ : capture variable environment
r is either a value or Ω (error)

Filters (by example)

	Jaql expression	Filter
Field access	$e\,.\ell$	$e\,;\{\ell : x\,,..\}$=>x
Conditional	`if` e_1 `then` e_2 `else` e_3	e_1 ; ‘`true`=>e_2 | ‘`false`=>e_3
Filter	`filter each x` `with x.size < 50`	`let` X = ‘`nil` => ‘`nil` |(x,xs)=> `if` x`.size < 50` `then` $(x\,,X\;xs)$ `else` $X\;xs$

Transform	`transform each x`
	`with`
	`{x.*,age:x.age+1}`
		`let` X = ‘`nil` => ‘`nil`
		|(`{"age"` : i=>$i+1$, ..}, y=>$X\;y$)

Typing filter application

"Evaluate the filter on the *type* of its argument"

Typing filter application

"Evaluate the filter on the *type* of its argument"

Definition (Type inference)

$$\Gamma \,;\, \Delta \,;\, M \vdash_{fil} f(t) : s$$

Γ capture variable environment

Δ recursion variable environment

M memoization environment (for recursive types)

$$\frac{\Gamma \cup t/p \,;\, \Delta \,;\, M \vdash_{fil} f(t) : s}{\Gamma \,;\, \Delta \,;\, M \vdash_{fil} p\texttt{=>}f(t) : s}\; t \leq \wr p \int \qquad \frac{i = 1,2 \quad \Gamma \,;\, \Delta \,;\, M \vdash_{fil} f_i(t) : s_i}{\Gamma \,;\, \Delta \,;\, M \vdash_{fil} f_1 \,|\, f_2(t) : s_1 | s_2}$$

$$\frac{\Gamma \,;\, \Delta, (X \mapsto f) \,;\, M, ((X, t) \mapsto T) \vdash_{fil} f(t) : s}{\Gamma \,;\, \Delta \,;\, M \vdash_{fil} (\texttt{let } X = f)(t) : \mu T.s}\; T \text{ fresh}$$

$$\frac{}{\Gamma \,;\, \Delta \,;\, M \vdash_{fil} (X\ a)(s) : T}\; \begin{array}{c} t = \texttt{type}(\Gamma, a) \\ ((X, t) \mapsto T) \in M \end{array}$$

...

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$\mathtt{let}\ \ Y = \text{`}\mathtt{nil} \ \texttt{=>}\text{`}\mathtt{nil}$ $\quad\quad\quad\quad \mu T.\text{`}\mathtt{nil}$

$\mathtt{|}(\{\texttt{"age"} : i\texttt{=>}i + 1\,,\ ..\}, y\texttt{=>}Y\ y)$ $\quad \big|\big(\big\{\texttt{"name"}:\mathtt{string}, \texttt{"age"}:\mathtt{int}\big\}, T\big)$

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$\texttt{let } Y = \texttt{`nil => `nil} \qquad \mu T.\texttt{`nil}$

$\texttt{|(\{"age"} : i \texttt{=>} i + 1 \texttt{, ..\}}, y \texttt{=>} Y\ y) \qquad |(\{\texttt{"name"}:\texttt{string}, \texttt{"age"}:\texttt{int}\}, T)$

Environments Output type

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$\texttt{let}\ Y = \texttt{`nil => `nil} \ |\ (\{\texttt{"age"} : i \texttt{=>} i+1\ ,\ ..\}, y \texttt{=>} Y\ y)$

$\mu T.\texttt{`nil} \ |\ (\{\texttt{"name"}:\texttt{string}, \texttt{"age"}:\texttt{int}\}, T)$

Environments

Output type

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

let Y = `nil =>`nil	$\mu T.$`nil
$\mid(\{$"age" $: i \texttt{=>} i+1, ..\}, y \texttt{=>} Y\ y)$	$\mid(\{$"name":string, "age":int$\}, T)$

Environments

$Y(T) \mapsto U$

Output type

$\mu U.$

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$\mathtt{let}\ \ Y = \text{`}\mathtt{nil}\ \ \mathtt{=>}\text{`}\mathtt{nil}$ $\qquad \mu T.\text{`}\mathtt{nil}$

$|\ (\{\mathtt{"age"} : i \mathtt{=>} i + 1\,,\ ..\}, y \mathtt{=>} Y\ \ y)$ $\qquad |(\{\mathtt{"name"}:\mathtt{string}, \mathtt{"age"}:\mathtt{int}\}, T)$

Environments

$Y(T) \mapsto U$

Output type

$\mu U. \qquad |$

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$\texttt{let}\ \ Y = \texttt{`nil => `nil}$ $\qquad\qquad\qquad\qquad$ $\mu T.\texttt{`nil}$

$\texttt{|}(\{\texttt{"age"} : i\texttt{=>}i + 1\, , \ldots\}, y\texttt{=>}Y\ y)$ $\quad$ $|(\{\texttt{"name"}:\texttt{string}, \texttt{"age"}:\texttt{int}\}, T)$

Environments

$Y(T) \mapsto U$

Output type

$\mu U.\texttt{`nil}|$

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$$\texttt{let}\ \ Y = \texttt{`nil}\ \texttt{=>`nil} \qquad\qquad \mu T.\texttt{`nil}$$

$$|\,(\{\texttt{"age"} : i \texttt{=>} i + 1\ , \ ..\}, y \texttt{=>} Y\ y) \quad |(\{\texttt{"name"}:\texttt{string}, \texttt{"age"}:\texttt{int}\}, T)$$

Environments

$Y(T) \mapsto U$

Output type

$\mu U.\texttt{`nil}|(\qquad , \quad)$

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$\texttt{let}\ \ Y = \texttt{`nil => `nil}$ $\qquad\qquad\qquad \mu T.\texttt{`nil}$

$\texttt{|}(\{\texttt{"age"} : i\texttt{=>}i + 1\ ,\ ..\}, y\texttt{=>}Y\ y)$ $\quad \texttt{|}(\{\texttt{"name"}:\texttt{string}, \texttt{"age"}:\texttt{int}\}, T)$

Environments

$Y(T) \mapsto U$

Output type

$\mu U.\texttt{`nil|}(\{\texttt{"name"}:\texttt{string},\ \texttt{"age"}:\quad\}, \quad)$

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$\texttt{let}\ \ Y = \texttt{`nil}\ \Rightarrow \texttt{`nil}$ $\qquad \mu T.\texttt{`nil}$

$\texttt{|}(\{\texttt{"age"} : i \Rightarrow i + 1\ ,\ ..\}, y \Rightarrow Y\ y)$ $\qquad |(\{\texttt{"name"}:\texttt{string}, \texttt{"age"}:\texttt{int}\}, T)$

Environments

$Y(T) \mapsto U$

$i \mapsto \texttt{int}$

Output type

$\mu U.\texttt{`nil}|(\{\texttt{"name"}:\texttt{string},\ \texttt{"age"}:\texttt{int}\},\quad)$

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

let Y = `nil =>`nil $\mu T.$`nil

| ({"age" : i=>$i+1$, ..}, y=> Y y) |({"name":string, "age":int}, T)

Environments

$Y(T) \mapsto U$

$i \mapsto$ int

$y \mapsto T$

Output type

$\mu U.$`nil|({"name":string, "age":int},)

Typing filter application (example)

Typing the application of

```
transform each x with { x.*, age: x.age+1 }
```

to a value of type

```
[ {name:string, age:int}* ]
```

$\texttt{let}\ \ Y = \texttt{`nil => `nil}$ $\qquad\qquad$ $\mu T.\texttt{`nil}$

$\texttt{|(\{"age"}: i \texttt{=>} i+1\texttt{, ..\}}, y \texttt{=>} Y\ y)$ $\qquad$ $|(\{\texttt{"name":string}, \texttt{"age":int}\}, T)$

Environments

$Y(T) \mapsto U$

$i \mapsto \texttt{int}$

$y \mapsto T$

Output type

$\mu U.\texttt{`nil}|(\{\texttt{"name":string}, \texttt{"age":int}\}, U)$

Typing problem : Termination

What about :

$$\texttt{let } Y = \texttt{`nil} \;\texttt{=>}\texttt{`nil} \;\;|\; (x,y) \texttt{=>} Y(x,(x,y))$$

applied to $\mu T.\, \texttt{`nil} | (\texttt{int}, T)$

Typing problem : Termination

What about :

$$\texttt{let}\ \ Y = \text{`}\texttt{nil}\ \texttt{=>}\text{`}\texttt{nil} \quad \mid \mathbf{(}x,y\mathbf{)}\texttt{=>}Y(x,(x,y)) \qquad \text{applied to} \quad \mu T.\ \text{`}\texttt{nil}\mathbf{|}\mathbf{(}\texttt{int},T\mathbf{)}$$

$Y(T) \mapsto U_1$

Typing problem : Termination

What about :

$$\begin{array}{l}\texttt{let } Y = \texttt{`nil => `nil} \\ \quad \texttt{|}(x,y)\texttt{=>} Y(x,(x,y))\end{array} \quad \text{applied to} \quad \mu T.\ \texttt{`nil|}(\texttt{int}, T)$$

$Y(T) \mapsto U_1$

$Y((\texttt{int},(\texttt{int}, T))) \mapsto U_2$

$Y((\texttt{int},(\texttt{int},(\texttt{int},(\texttt{int}, T))))) \mapsto U_3$

$\vdots$

Typing problem : Termination

What about :

$$\begin{array}{l}\texttt{let}\ \ Y = \texttt{`nil} \ \texttt{=>`nil} \\ \ \mid (x,y)\texttt{=>}\, Y(x,(x,y))\end{array} \quad \text{applied to} \quad \mu T.\ \texttt{`nil} \mid (\texttt{int}, T)$$

$Y(T) \mapsto U_1$
$Y((\texttt{int},(\texttt{int},T))) \mapsto U_2$
$Y((\texttt{int},(\texttt{int},(\texttt{int},(\texttt{int},T))))) \mapsto U_3$
$\vdots$

How to refuse such ill-founded filters ?

Typing problem : Termination

What about :

$$\texttt{let } Y = \texttt{`nil} \Rightarrow \texttt{`nil} \mid (x,y) \Rightarrow Y(x,(x,y)) \quad \text{applied to} \quad \mu T.\, \texttt{`nil} | (\texttt{int}, T)$$

$Y(T) \mapsto U_1$
$Y((\texttt{int},(\texttt{int},T))) \mapsto U_2$
$Y((\texttt{int},(\texttt{int},(\texttt{int},(\texttt{int},T))))) \mapsto U_3$
$\vdots$

How to refuse such ill-founded filters ?

1. Assign an identifier to each (term) variable : $x \mapsto i_1$, $y \mapsto i_2$,

Typing problem : Termination

What about :

$$\texttt{let } Y = \texttt{`nil} \texttt{ =>`nil} \qquad \text{applied to} \quad \mu T.\ \texttt{`nil}|(\texttt{int}, T)$$
$$|(x,y)\texttt{=>} Y(x,(x,y))$$

$Y(T) \mapsto U_1$
$Y((\texttt{int},(\texttt{int},T))) \mapsto U_2$
$Y((\texttt{int},(\texttt{int},(\texttt{int},(\texttt{int},T))))) \mapsto U_3$
$\vdots$

How to refuse such ill-founded filters ?

1. Assign an identifier to each (term) variable : $x \mapsto i_1$, $y \mapsto i_2$,
2. For each recursive call, build an abstract value : $(i_1,(i_1,i_2))$

Typing problem : Termination

What about :

```
let Y = `nil => `nil
     | (x,y) => Y(x, (x, y))
```

applied to $\mu T.\ \texttt{`nil}|(\texttt{int}, T)$

$Y(T) \mapsto U_1$
$Y((\texttt{int},(\texttt{int}, T))) \mapsto U_2$
$Y((\texttt{int},(\texttt{int},(\texttt{int},(\texttt{int}, T))))) \mapsto U_3$
$\vdots$

How to refuse such ill-founded filters ?

1. Assign an identifier to each (term) variable : $x \mapsto i_1$, $y \mapsto i_2$,
2. For each recursive call, build an abstract value : $(i_1, (i_1, i_2))$
3. Apply the filter to the abstract values. Variables must be bound to exactly one identifier : $x \mapsto i_1$, $y \mapsto (i_1, i_2)$

Notable results

1. Type safety (of course !)
   If $\varnothing\,⨾\,\varnothing\,⨾\,\varnothing \vdash_{fil} f(t) : s$, then $\forall v : t,\ \varnothing\,;\varnothing \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$
   (in particular, $r \neq \Omega$)
2. Precise typing of record expressions

Notable results

1. Type safety (of course !)
   If $\varnothing\,;\,\varnothing\,;\,\varnothing \vdash_{fil} f(t) : s$, then $\forall v : t$, $\varnothing\,;\varnothing \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$
   (in particular, $r \neq \Omega$)
2. Precise typing of record expressions
3. Encode and type all Jaql and Pig/Latin operators as well as XML Schemas and downward XPath

Notable results

1. Type safety (of course !)
   If $\varnothing\,;\varnothing\,;\varnothing \vdash_{fil} f(t) : s$, then $\forall v : t,\ \varnothing\,;\varnothing \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$ (in particular, $r \neq \Omega$)
2. Precise typing of record expressions
3. Encode and type all Jaql and Pig/Latin operators as well as XML Schemas and downward XPath
4. Typechecking is EXPTIME (semantic subtyping is EXPTIME complete already)

Notable results

1. Type safety (of course !)
   If $\varnothing\,{}^{\circ}_{9}\,\varnothing\,{}^{\circ}_{9}\,\varnothing \vdash_{fil} f(t) : s$, then $\forall v : t,\ \varnothing\,;\!\varnothing \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$ (in particular, $r \neq \Omega$)
2. Precise typing of record expressions
3. Encode and type all Jaql and Pig/Latin operators as well as XML Schemas and downward XPath
4. Typechecking is EXPTIME (semantic subtyping is EXPTIME complete already)
5. Arbitrary filters are Turing-complete

Notable results

1. Type safety (of course !)
   If $\varnothing ⨾ \varnothing ⨾ \varnothing \vdash_{fil} f(t) : s$, then $\forall v : t$, $\varnothing ; \varnothing \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$ (in particular, $r \neq \Omega$)
2. Precise typing of record expressions
3. Encode and type all Jaql and Pig/Latin operators as well as XML Schemas and downward XPath
4. Typechecking is EXPTIME (semantic subtyping is EXPTIME complete already)
5. Arbitrary filters are Turing-complete
6. Typeable filters are more-expressive than Top-down tree transducers with regular look-ahead

Notable results

1. Type safety (of course !)
   If $\varnothing\,;\varnothing\,;\varnothing \vdash_{fil} f(t) : s$, then $\forall v : t,\ \varnothing\,;\varnothing \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$ (in particular, $r \neq \Omega$)
2. Precise typing of record expressions
3. Encode and type all Jaql and Pig/Latin operators as well as XML Schemas and downward XPath
4. Typechecking is EXPTIME (semantic subtyping is EXPTIME complete already)
5. Arbitrary filters are Turing-complete
6. Typeable filters are more-expressive than Top-down tree transducers with regular look-ahead
7. Sound (approximate) typing of non structural operators (`group_by`, `join`, `order_by`, ...)

Some thoughts…

- DB community always comes up with interesting languages : SQL, XML query languages, NoSQL languages, RDF querying…
- Almost never a decent "safety oriented" static analysis
- Filter as type level combinators allows us to balance :
  - expressivity
  - decidability
  - precision
  - exotic use of polymorphism and subtyping

  with some costs
  - Not for higher-order languages
  - Modularity

Summary, future work

Summary :

1. Precise JSON schema via regexp type via semantic subtyping
2. Expressive calculus of combinators to encode iterators
3. Precise typing of filter application

⇒ framework for ensuring type-safety of NoSQL programs

Future work :

1. Relax some conditions on static analysis (allows one to express `count`, `average`, `sum` and other numerical agregate functions)

   $$\texttt{let } X = (c, \texttt{`nil}) \texttt{=>} c$$
   $$\quad | (c, (x, xs)) \texttt{=>} X\ (c + x, xs)$$

2. Implementation effort to integrate in the Jaql framework
3. Study connections between filters and the actual compilation scheme (MapReduce)