Static and Dynamic Semantics of NoSQL Languages

POPL 2013, Rome, Jan. 23-25 2013

Véronique Benzaken\textsuperscript{1}  Giuseppe Castagna\textsuperscript{2}
\textbf{Kim Nguyễn\textsuperscript{1}}  Jérôme Siméon\textsuperscript{3}

\textsuperscript{1} LRI, Université Paris-Sud, Orsay, France
\textsuperscript{2} CNRS, PPS, Univ Paris Diderot, Paris, France
\textsuperscript{3} IBM Watson Research, Hawthorne, NY, USA
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 aggregation of intermediary results (Reduce)
- Data-intensive applications
SQL (and the Relational DBMS) are not good for everything.

NoSQL is a 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 aggregation 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) : Jql, Pig/Latin, Sawzall, Unql, ...
- No formal semantics \(\Rightarrow\) hard to reason about the code
- Weak notion of schema (data types) \(\Rightarrow\) 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 $\Rightarrow$ hard to reason about the code
- Weak notion of schema (data types) $\Rightarrow$ 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
cmpls = [ {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
depths = [{depnum:154, name:"HR", size:40}, ... ];
//sequence of employee records
empls = [{name:"Kim", depid:"210", salary:1000}, ... ]
union(
    depths -> 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?
∀α.[α] → [α] → [α] 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?
∀α.[α] → [α] → [α] seems too restrictive...
[any] → [any] → [any] seems too imprecise...
Jaql in a nutshell

Data-model is JavaScript Object Notation (JSON)

// sequence of department records
degrts = [ { 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?

∀α. [α] → [α] → [α] seems too restrictive...

[any] → [any] → [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`?

```typescript
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?

```typescript
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?
Definition (Types)

\[ t ::= \text{int} | \text{string} | \ldots \quad \text{(basic types)} \]
\[ \text{‘nil} | 42 | \ldots \quad \text{(singleton types)} \]
\[ (t,t) \quad \text{(products)} \]
\[ \{\ell:t,\ldots,\ell:t\} \quad \text{(closed records)} \]
\[ \{\ell:t,\ldots,\ell:t,\ldots\} \quad \text{(open records)} \]
\[ t|t \quad \text{(union types)} \]
\[ t\&t \quad \text{(intersection types)} \]
\[ \neg t \quad \text{(negation type)} \]
\[ \text{empty} \quad \text{(empty type)} \]
\[ \text{any} \quad \text{(any type)} \]
\[ \mu T.t \quad \text{(recursive types)} \]
\[ T \quad \text{(recursion variable)} \]

\[ \mu T.(\text{‘nil} | (\{"name": \text{string}, \ldots\}, T) ) \equiv \]
\[ [\{\text{name}: \text{string}, \ldots\}*] \]
Definition (Semantic subtyping)

\[ s \leq t \iff [s] \subseteq [t] \]

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

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

Definition (Basic expressions)

\[ e ::= c \quad \text{(constants)} \]
\[ | \quad x \quad \text{(variables)} \]
\[ | \quad (e, e) \quad \text{(pairs)} \]
\[ | \quad \{e:e, ..., e:e\} \quad \text{(records)} \]
\[ | \quad e + e \quad \text{(record concatenation)} \]
\[ | \quad e \ell \quad \text{(field deletion)} \]
\[ | \quad \text{op}(e, ..., e) \quad \text{(built-in operators)} \]
\[ | \quad f \ e \quad \text{(filter application)} \]

Example

\{"age":30, "name":"Kim"\} + \{"age":31\} \leadsto \{"age":31, "name":"Kim"\}

\{"age":30, "name":"Kim"\} \setminus "name" \leadsto \{"age":30\}

\{strconcat("a", "ge"):30\} \leadsto \{"age":30\}
Basic expression typing

[VARS] \[ \Gamma \vdash x : \Gamma(x) \]

[CONSTANT] \[ \Gamma \vdash c : c \]

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

[FOREIGN] \[ \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash \text{op}(e_1, \ldots, e_n) : \text{type}((\Gamma, x_1 : t_1, \ldots, x_n : t_n), \text{op}(x_1, \ldots, x_n))} \]

[RCD-FIN] \[ \frac{\Gamma \vdash e : \ell_1 \mid \cdots \mid \ell_n \quad \Gamma \vdash e' : t}{\Gamma \vdash \{e : e'\} : \{\ell_1 : t\} \mid \cdots \mid \{\ell_n : t\}} \]

[RCD-INF] \[ \frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t'}{\Gamma \vdash \{e : e'\} : \{..\}} \]
\[ t \leq \text{string} \]
\[ t \text{ is infinite} \]
**Filters**

Definition (Filters)

\[ f ::= e \quad (\text{expression}) \]
\[ p => f \quad (\text{pattern}) \]
\[ f | f \quad (\text{union}) \]
\[ f ; f \quad (\text{composition}) \]
\[ (f , f) \quad (\text{product}) \]
\[ \{ l : f , \ldots , l : f , \ldots \} \quad (\text{record}) \]
\[ \text{let } X = f \quad (\text{rec. definition}) \]
\[ X a \quad (\text{guarded rec.}) \]

Definition ((Big-step) semantics of filters)

\[ \delta ; \gamma \vdash_{\text{eval}} f(v) \rightsquigarrow r \]

\( \delta \): recursion variable environment
\( \gamma \): capture variable environment
\( r \) is either a value or \( \Omega \) (error)
## Filters (by example)

<table>
<thead>
<tr>
<th>Jaql expression</th>
<th>Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Field access</strong></td>
<td>$e \cdot ℓ$</td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>if $e_1$ then $e_2$ else $e_3$</td>
</tr>
<tr>
<td><strong>Filter</strong></td>
<td>filter each $x$ with $x\cdot\text{size} &lt; 50$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Transform</strong></td>
<td>transform each $x$ with ${ x\cdot\ast, \text{age}: x\cdot\text{age}+1 }$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)

\[ \Gamma ; \Delta ; M \vdash_{fil} f(t) : s \]

\( \Gamma \) capture variable environment

\( \Delta \) recursion variable environment

\( M \) memoization environment (for recursive types)

\[ \Gamma \cup t/p ; \Delta ; M \vdash_{fil} f(t) : s \quad t \leq \gamma p \]

\[ \Gamma ; \Delta ; M \vdash_{fil} p \Rightarrow f(t) : s \]

\[ i = 1, 2 \quad \Gamma ; \Delta ; M \vdash_{fil} f_i(t) : s_i \]

\[ \Gamma ; \Delta ; M \vdash_{fil} f_1 | f_2(t) : s_1 | s_2 \]

\[ \Gamma ; \Delta, (X \mapsto f); M, ((X, t) \mapsto T) \vdash_{fil} f(t) : s \quad T \text{ fresh} \]

\[ \Gamma ; \Delta ; M \vdash_{fil} (\text{let } X = f)(t) : \mu T.s \]

\[ \Gamma ; \Delta ; M \vdash_{fil} (X \ a)(s) : T \quad \text{if } t = \text{type}(\Gamma, a) \]

\[ ((X, t) \mapsto T) \in M \]

\[ \ldots \]
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}* ]

let Y = 'nil =>'nil  \( \mu T.\text{nil} \)
|({"age": i=>i+1, ..}, y=>Y y) |({"name":string,"age":int}, T)
Typing filter application (example)

Typing the application of

transform each \( x \) with \{ \( x.* \), age: \( x.\text{age}+1 \) \}

to a value of type

\[ \{\text{name: string}, \text{age: int}\}^* \]

\[
\text{let } Y = '\text{nil}' \Rightarrow '\text{nil}' \\
\mu T.'\text{nil}' \\
\mid (\{"age": i\Rightarrow i+1, \ldots\}, y\Rightarrow Y \ y) \\
\mid (\{"name": \text{string}, "age": \text{int}\}, T)
\]

<table>
<thead>
<tr>
<th>Environments</th>
<th>Output type</th>
</tr>
</thead>
</table>

Type inference
Typing filter application (example)

Typing the application of

\[ \text{transform each } x \text{ with } \{ x.*, \text{ age: } x.\text{age}+1 \} \]

to a value of type

\[ [ \{ \text{name:}\text{string, age:}\text{int}\}^* ] \]

\[ \text{let } Y = '\text{nil' \to 'nil' \mu T.'nil} \]
\[ |(\{\text{"age":}\text{i \to i + 1, \ldots}\}, y \to Y y) |(\{\text{"name":}\text{string, "age":}\text{int}\}, T) \]

Environments \hspace{1cm} Output type
Typing filter application (example)

Typing the application of
transform each x with \{ x.*, age: x.age+1 \}
to a value of type
\[ \{ \text{name: string, age: int}\}^* \]

\[
\text{let } Y = '\text{nil } \Rightarrow '\text{nil } \\
\mu T. '\text{nil } \\
| (\{"age": i \mapsto i + 1, \ldots\}, y \mapsto Y y) \\
| (\{"name": \text{string}, "age": \text{int}\}, T)
\]

Environments
\[
Y(T) \mapsto U
\]

Output type
\[
\mu U.
\]
Typing filter application (example)

Typing the application of

transform each \( x \) with \( \{ x.*, \text{age: } x.\text{age}+1 \} \)

to a value of type

\[ \{ \{ \text{name: string, age: int} \} \} \]

\[
\text{let } Y = \text{\'nil =>\'nil}
| (\{\text{\"name\": string, \"age\": int} \},,)
| ((\{\text{\"name\": string, \"age\": int}\}, T))
\]

Environments

\( Y(T) \mapsto U \)

Output type

\( \mu U \cdot \)
Typing filter application (example)

Typing the application of

\[
\text{transform each } x \text{ with } \{ x.*, \text{age: } x.\text{age}+1 \}
\]
to a value of type

\[
[ \{\text{name:string, age:int}\}*] 
\]

let \( Y = \text{\'nil =>\'nil} \quad \mu T.\text{\'nil} \)

\[
|\{\text{\"age\": } i=>i+1, \ldots, y=>Y \ y\} |(\{\text{\"name\":string, \"age\":int}\}, T) \]

Environments

\( Y(T) \mapsto U \)

Output type

\( \mu U.\text{\'nil} | \)

Type inference
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 \( Y = '\text{nil} \Rightarrow '\text{nil} \)
| \({"age": i => i + 1, \ldots}, y => Y y\) \| \(({"name": string, "age": int}, T)\)

Environments

\( Y(T) \mapsto U \)  \hspace{1cm} \text{Output type}
\( \mu U. '\text{nil}\|(') \)
Typing filter application (example)

Typing the application of

\[ \text{transform each } x \text{ with } \{ x.*, \text{ age: } x.\text{age}+1 \} \]

to a value of type

\[ [ \{ \text{name: string, age: int}\}*. \] \]

let \( Y = \text{"nil =>"nil} \) \( Y \Rightarrow Y \) 
\( \mu T.\text{"nil} \) 
\( |(|\{\text{name: string, age: int}\}, T)\) 

Environments
\( Y(T) \mapsto U \)

Output type
\( \mu U.\text{"nil}|(\{\text{name: string, age: } \}) \)
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 \( Y = \text{\textquote{\textquotesingle}nil => \text{\textquotesingle}nil} \)
\mu T.\text{\textquotesingle}nil

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

Environments

\( Y(T) \mapsto U \)

\( i \mapsto \text{int} \)

Output type

\mu U.\text{\textquotesingle}nil\|((\{"name":string, "age":int\}, )\)
Typing the application of
transform each \( x \) with \( \{ x.*, \text{age: } x.\text{age}+1 \} \)
to a value of type
\[
[ \{ \text{name: string, age: int}\}* \]
\]

let \( Y = \text{`nil =>`nil} \)
\[
| (\{"age": i=>i+1, \ldots\}, y=>Y \ y) | (\{"name": string, "age": int\}, T)
\]

Environments
\[
Y(T) \mapsto U
\]
\[
i \mapsto \text{int}
\]
\[
y \mapsto T
\]

Output type
\[
\mu U.\text{`nil} | (\{"name": string, "age": int\}, )
\]

Type inference
Typing the application of
transform each x with \{ x.*, age: x.age+1 \}
to a value of type
\[ \{ \text{name:}\text{string}, \text{age:}\text{int}\}* \]
Typing problem: Termination

What about:

\[
\text{let } Y = '\text{nil} \Rightarrow '\text{nil} \\
| (x,y) \mapsto Y(x,(x,y))
\]

applied to \( \mu T. '\text{nil}|(\text{int}, T) \)
What about:

\[
\text{let } Y = 'nil \Rightarrow 'nil \\
| (x, y) => Y(x, (x, y))
\]

applied to \( \mu T. 'nil | (\text{int, } T) \)

\( Y(T) \mapsto U_1 \)
Typing problem : Termination

What about:

\[
\text{let } Y = \text{"nil \Rightarrow \text{\textbf{\textcolor{red}{nil}}} \} \Rightarrow \} \} \Rightarrow \) \}
| (x,y) \Rightarrow Y(x, (x, y))
\]

applied to \( \mu \ T. \text{"nil\}((\text{int}, T)\)} \)

\[
Y(T) \mapsto U_1
Y((\text{int}, (\text{int}, T))) \mapsto U_2
Y((\text{int}, (\text{int}, (\text{int}, (\text{int}, T)))))) \mapsto U_3
\]

\[
\text{\ldots}
\]
Typing problem : Termination

What about :

\[
\text{let } Y = '\text{nil} \Rightarrow '\text{nil} \\
| (x, y) \Rightarrow Y(x, (x, y))
\]

applied to \( \mu T. '\text{nil}|(\text{int}, T) \)

\[
Y(T) \mapsto U_1 \\
Y((\text{int}, (\text{int}, T))) \mapsto U_2 \\
Y((\text{int}, (\text{int}, (\text{int}, (\text{int}, T))))) \mapsto U_3 \\
\vdots
\]

How to refuse such ill-founded filters?
Typing problem: Termination

What about:

\[
\text{let } Y = \text{'nil } \Rightarrow \text{'nil} \\
\mid (x,y) \Rightarrow Y(x,(x,y)) \quad \text{applied to } \mu T. \text{'nil} \mid (\text{int}, T)
\]

\[
Y(T) \mapsto U_1 \\
Y((\text{int},(\text{int}, T))) \mapsto U_2 \\
Y((\text{int},(\text{int},(\text{int},(\text{int}, T))))) \mapsto U_3
\]

\[\vdots\]

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 :

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

applied to $\mu T. 'nil | (\text{int}, T)$

$Y(T) \mapsto U_1$

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

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

::

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:

\[
\text{let } Y = \text{‘nil} \Rightarrow \text{‘nil} \\
| (x, y) \Rightarrow Y(x, (x, y))
\]

applied to \( \mu T. \text{‘nil}|(\text{int}, T) \)

\[
Y(T) \mapsto U_1 \\
Y(\text{int},(\text{int}, T))) \mapsto U_2 \\
Y(((\text{int},(\text{int},(\text{int},T))))) \mapsto U_3 \\
\]

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 $\emptyset ; \emptyset ; \emptyset \vdash_{fil} f(t) : s$, then $\forall v : t$, $\emptyset ; \emptyset \vdash_{eval} f(v) \rightsquigarrow r$ implies $r : s$
   (in particular, $r \neq \Omega$)

2. Precise typing of record expressions
Notable results

1. Type safety (of course!)
   If \( \emptyset ; \emptyset ; \emptyset \vdash_{fil} f(t) : s \), then \( \forall v : t, \emptyset ; \emptyset \vdash_{eval} f(v) \leadsto r \) 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

Summary and future work
Notable results

1. Type safety (of course!)
   If $\emptyset;\emptyset;\emptyset \vdash_{fil} f(t) : s$, then $\forall v : t, \emptyset;\emptyset \vdash_{eval} f(v) \leadsto r$ 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!)
   \[ \emptyset;\emptyset;\emptyset; \vdash_{fil} f(t) : s, \text{ then } \forall v : t, \emptyset;\emptyset; \vdash_{eval} f(v) \Rightarrow r \text{ implies } r : s \]
   (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 \( \text{EXPTIME} \) (semantic subtyping is \( \text{EXPTIME} \) complete already)

5. Arbitrary filters are Turing-complete

Summary and future work
Notable results

1. Type safety (of course!)
   If $\emptyset;\emptyset;\emptyset \vdash_{fil} f(t) : s$, then $\forall v : t, \emptyset;\emptyset \vdash_{eval} f(v) \leadsto r$ 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

Summary and future work
Notable results

1. Type safety (of course!)
   If $\emptyset; \emptyset; \emptyset \vdash_{fil} f(t) : s$, then $\forall v : t, \emptyset; \emptyset \vdash_{eval} f(v) \sim r$ 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, ...)

Summary and future work
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 aggregate functions)

   let \( X = (c, \text{'nil}) \Rightarrow c \)
   \((c, (x, xs)) \Rightarrow X (c + x, xs)\)

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