Une sémantique statique pour MongoDB

Adrien Husson

4 janvier 2014

Contexte

MongoDB : Une base de données *non*-relationnelle

- Données et requêtes : syntaxe JSON
- Tables hétérogènes, structures imbriquées
- Pas de sémantique formelle, pas de typage

Contexte

MongoDB : Une base de données *non*-relationnelle

- **Données et requêtes : syntaxe JSON**
- Tables hétérogènes, structures imbriquées
- Pas de sémantique formelle, pas de typage

Syntaxe JSON

```
{
 dept_id: 100,
 scores: [12, 17, null, 14],
 name: {
  first: "John",
  middle: "Thomas",
  last: "Smith"
 }
}
```

Contexte

MongoDB : Une base de données *non*-relationnelle

- Données et requêtes : syntaxe JSON
- **Tables hétérogènes, structures imbriquées**
- Pas de sémantique formelle, pas de typage

Contenu possible d'une table

```
[
 {dept_id: 23},
 {dept_id: true, scores: [10, "good",{}]},
 {name: {first: "John", last: null}, scores: [0]}
]
```

Contexte

MongoDB : Une base de données *non*-relationnelle

- Données et requêtes : syntaxe JSON
- Tables hétérogènes, structures imbriquées
- **Pas de sémantique formelle, pas de typage**

Pas de modèle d'éxécution / typage

- Pas de *schéma* (point de vue abstrait)
- Idée de ce que va renvoyer une requête sans l'éxécuter ?

Contexte

Résultats

- Traduction de MongoDB[1] vers un langage typé muni d'une sémantique formelle
- Traduction implémentée
- Typage sans annotation de requêtes MongoDB
  - Détecter de requêtes erronnées
  - Vérifier l'adhérence à un schéma

[1] Traduction des recherches, mises à jour et projections.

En cours : opérations map-reduce.

Exemple de requête MongoDB

update(*Find*, *Update*)

Find `$or: [{address: "Dorms"}, {$exists: {name: true}}]`
Trouve les éléments qui satisfont **soit** :

- Le champ `address` **est** ou **est une liste contenant** `"Dorms"`.
- Le champ `name` existe.

Update `$inc: {scores: 2}`
Les éléments trouvés voient leur champ `scores` :

- incrémenté de 2 *si c'est un nombre*,
- fixé à 2, *s'il n'existe pas*,
- erreur, *sinon*

Le préfixe $ indique un opérateur MongoDB

But : typer les éxécutions de requêtes MongoDB

- Obtenir un schéma (type) des données capables de satisfaire un ensemble de requêtes.
- Vérifier qu'un schéma est respecté après une mise à jour
- Détecter des erreurs possibles
  e.g. `$inc: {a: 2}` dans un contexte où a peut ne pas être un nombre.

Typer la partie *Find*

Condition 1 = `[{address: string}]`
Condition 2 = `[{address: list, name: string}]`

Quel est le type d'une liste dont les éléments satisfont *Condition 1* ou *Condition 2* ?

$[\alpha]$ trop restrictif, `[any]` pas assez précis

Avec des opérations ensemblistes, on aurait :
`[({address: string} | {address: list, name: string})]`

Typer avec des langages d'arbres (1/2)

[POPL13, Benzaken & al.]

t : :=	int | string | ...	(base)
	| nil | 42 | ...	(singleton)
	| (t, t)	(produit)
	| $\{\ell : t, \ldots, \ell : t\}$	(record fermé)
	| $\{\ell : t, \ldots, \ell : t, ..\}$	(record ouvert)
	| $t \mid t$	(union)
	| $t \& t$	(intersection)
	| $\neg t$	(négation)
	| `empty`	(vide)
	| `any`	(toutes les valeurs)
	| type rec $T = t$	(type récursif)
	| T	(variable de récursion)

Chaque type est un langage d'arbre régulier
On peut identifier un valeur v et son type singleton $\{v\}$.

Typer avec des langages d'arbres (2/2)

[POPL13, Benzaken & al.]

Type d'une liste

type rec $T = (\texttt{string} \mid \texttt{null}, T) \mid \text{nil}$

Syntaxe regexp

`[(string | null)*]`

Expressions régulières arbitraires : `[char+ (int|bool)?]`

Typer la partie *Find*, suite.

Q = `$or: [{address: "Dorms"}, {$exists: {name: true}}]`

$[\![Q]\!]$ = {address : ("Dorms" | [any* "Dorms" any*]),..}
| {name : any,..}

Cas général

Récursion sur la structure de la recherche

Typer la partie *Update*

Besoin d'un langage de transformation de données

- Formellement spécifié
- Typable sans annotations

`<data>` $\rightsquigarrow$ Filtre 1 $\rightsquigarrow$ ... $\rightsquigarrow$ Filtre N $\rightsquigarrow$ `<output>`

Filtres (1/3)

[POPL13, Benzaken & al.]

- Sémantique formelle
- Typé par des langages d'arbres réguliers
- Orienté vers la transformation de données

$$
\begin{array}{lll}
\textbf{Expression } e ::= & c & \text{(constante)} \\
& \mid x & \text{(variable)} \\
& \mid (e, e) & \text{(paire)} \\
& \mid \{e : e, \ldots, e : e\} & \text{(record)} \\
& \mid e + e & \text{(concaténation de record)} \\
& \mid e \backslash \ell & \text{(suppression de champ)} \\
& \mid \mathrm{op}(e, \ldots, e) & \text{(opérateur)} \\
& \mid f\ e & \text{(application de filtre)}
\end{array}
$$

```
{id: 100, name: "Natural philosophy"} + {name: "Physics"}
⇝  {id: 100, name: "Physics"}

{id: 0, name: "Physique"} \ name  ⇝  {id: 100}
```

Filtres (2/3)

[POPL13, Benzaken & al.]

Filtre f : :=	e	(expression)
	$\mid o \Rightarrow f$	(pattern)
	$\mid f \mid f$	(union)
	$\mid f ; f$	(composition)
	$\mid (f, f)$	(produit)
	$\mid \{\ell : f, \ \ldots, \ \ell : f, ..\}$	(record)
	$\mid$ let rec X $= f$	(filtres récursif)
	$\mid$ X(a)	(appel récursif)

Le langage est muni d'une sémantique opérationnelle à grands pas.

Filtres (3/3)

[POPL13, Benzaken & al.]

Sémantique informelle

let rec $X =$

$\quad$ ({`size`: (`int` & $x \Rightarrow x+1 \mid \mathrm{x} \Rightarrow \mathrm{x}$),..}, $t \Rightarrow X(t)$)

$\quad \mid (h, t) \Rightarrow X(t)$

$\quad \mid \mathrm{nil} \Rightarrow \mathrm{nil}$

"Supprimer les éléments de l'entrée qui ne sont pas un record avec un champ `size`*. Pour les autres, si la valeur de* `size` *est un entier, l'augmenter de 1, sinon, ne rien faire."*

```
 [{id: 100, name: "Physics", size: "231"}, {name: "John Smith"},
{id: 98, name: "Math", size: π}]
⇝
[{id: 100, name: "Physics", size: "232"}, {id: 98, name: "Math",
size: π}]
```

Inférence de type

[POPL13, Benzaken & al.]

- Typage habituel pour les expressions
- On type l'*application* d'un filtre à un *type*.
  - Les filtres ne sont pas des objets de première classe
  - Les jugements ont la forme suivante (simplifiée) :

Exemple : Filtre paire

$$\frac{f_1(t_1) : s_1 \qquad f_2(t_2) : s_2}{(f_1, f_2)\ (t_1, t_2) : (s_1, s_2)}$$

Si l'application de f_i au type t_i renvoie un valeur de type s_i, l'application du filtre paire (f_1, f_2) au type paire (t_1, t_2) renvoie une valeur de type (s_1, s_2)

Typage d'une *Update*

MongoDB

`update(`$\underbrace{\texttt{\{scores: null\}}}_{Q}$`, `$\underbrace{\texttt{\{\$set: \{scores.\$: 0\}\}}}_{U}$`)`

Ici, `$` est une référence à l'index du premier élément dans `scores` égal à `null`.

Comportement

`{scores: [2,null,10,null,29]}` $\rightsquigarrow_{\texttt{update}(Q,U)}$
`{scores: [2,0,  10,null,29]}`

Traduction vers un filtre

$[\![U]\!] = [\![U]\!]([\![Q]\!]) =$ {scores : Find(`null` $\Rightarrow$ 0),..}

Le filtre Find(f) traverse une liste jusqu'à ce que l'application de f réussisse.

Type inféré

`{scores: [int* null? int*]}` $\vdash_{\texttt{update}(Q,U)}$ `{scores: [int*]}`

Conclusion + futur

Conclusion

- Une traduction de plus vers les filtres (Jaql et XML déjà partiellement faits)
- Implémentés : typage des filtres, traduction de MongoDB vers les filtres

Futur

- Créer une série de tests pour comparer le comportement de MongoDB à la sémantique de son encodage
- Autres moyens de garantir la correction de la traduction ?
- Site web pour que de sdéveloppeurs MongoDB puissent tester leurs requêtes
- Implémenter les types natifs MongoDB au niveau des filtres...
- ...pour achever la traduction de MongoDB au niveau des valeurs.

Typage d'une *Update* (bonus)

Une update est associée à une requête Q et a la forme suivante :

`path: {$operation: value}`

Dans le cas général

1. Convertir `{$operation: value}` vers un filtre f.
2. Traverse `chemin` (sans le jeter), construire un filtre le long de la structure.
3. Si le prochain noeud du chemin est $, utiliser le préfixe traversé jusqu'ici + la Q associée pour construire le prochain filtre.
4. En fin de chemin, appliquer f.

Translation of *Projection* (1/2)

Two types of projection : by exclusion and by inclusion.

Example for inclusion

$P =$ `{name.first: 1, name.middle: 1}`

```
{ ...,  name: {
          first: "John",
          middle: "Thomas"
          last: "Smith" }}
```

$\rightsquigarrow_P$

```
{name: {first: "John",
          middle: "Thomas" }}
```

If paths $p.p_1$ and $p.p_2$ are included, the filter must grab p_1 and p_2 at once.

Translation of *Projection* (2/2)

Example for inclusion

$P =$ {name.first: 1, name.middle: 1}

$[\![P]\!] = F_1 \mid \text{Filter}_{F_1}$

$F_1 = (\{\text{name} : \{\ F_2 \mid \text{Filter}_{F_2}\ \},..\} \mid x \Rightarrow x\backslash\text{name})\ ;$
$\quad x \Rightarrow \{\text{name}: \square\} \oplus_\square x$

$F_2 = (\{\text{first} : \text{Id},..\} \mid (x \Rightarrow x\backslash\text{first}))\ ;$
$\quad (\{\text{middle} : \text{Id},..\} \mid (x \Rightarrow x\backslash\text{middle}))\ ;$
$\quad x \Rightarrow \{\text{first} : \square, \text{middle} : \square\} \oplus_\square x$

$\square$ is a fresh atom and $r_1 \oplus_v r_2$ replaces the field of r_1 of value v with the value of the corresponding fields of r_2 (including non-existing fields, where the value is written $\perp$).

Filter_f goes through a list and only keeps elements for which f succeeds.

Translation of *Find* (1/2)

Let Q be a query, we translate it to a filter of the shape $[\![Q]\!]\&x \Rightarrow x$ where $[\![Q]\!]$ is a *type*.

Example :

```
Q = {history: {$elemMatch: {type: "Boarding school"}},
     grades.2: {$in: ["F","E"]}}
```

$[\![Q]\!]$ = {items : [any* {type : ["Boarding school"]$^?$,..} any*],..}
& {grades : [any any ["F"|"E"]$^?$ any*],..}

where for a type t, $[t]^? = \mathsf{t} \mid [\mathtt{any*}\ t\ \mathtt{any*}]$

Translation of *Find* (2/2)

Some other translations

$[\![\texttt{\$all:[}v_1\texttt{, ..., }v_n\texttt{]}]\!] = (\&_{i=1}^{n} v_i) \mid (\&_{i=1}^{n} \texttt{[any* }v_i\texttt{ any*]})$

- Either all v_i's are equal, and the matched value is equal to them
- Or the matched value is an array and it contains all v_i's

$[\![\texttt{\$ne:}\, v]\!] = \neg[v]^? = \neg v \;\&\; \neg\texttt{[any* }v\texttt{ any*]}$

- Matched value is not equal to v and is not an array containing v.

Cas particuliers MongoDB à respecter ?

Recherche + mise à jour

Find `a.b: {$elemMatch: {x: 1}}`

Update
```
$inc: {a.b.$.y: 2},
      $set: {a.c.1: false}
```

Recherche

- `a.b` : `{a: {b: 42}}` ou `{a: [... {b: 42} ...]}`
- `a: {$elemMatch: value}` :
  Trouver *value* : `a: [... value ...]`

Mise à jour

- `$inc` : incrémenter, `$set` : assigner
- `a.b.$.x` (*backreference*) : Trouver l'élément qui a fait réussir le `$elemMatch` et modifier son champ `x`.

Cas particuliers MongoDB à respecter ?

Recherche + mise à jour

Find `a.b: {$elemMatch: {x: 1}}`

Update
```
$inc: {a.b.$.y: 2},
      $set: {a.c.1: false}
```

Exemples

Sur l'entrée

```
{a:
 {b: [true, {x: 1}],
  c: []}}
```

On obtient

```
{a:
 {b: [true, {y: 2, x: 1},
  c: [null, false]}}
```

Cas particuliers MongoDB à respecter ?

Recherche + mise à jour

Find `a.b: {$elemMatch: {x: 1}}`

Update
```
$inc: {a.b.$.y: 2},
      $set: {a.c.1: false}
```

Exemples

Sur l'entrée

```
{a:
 {b: [true, {y: 1, x: 1}]}}
```

On obtient

```
{a:
 {b: [true, {y: 3, x: 1}],
  c: {1: false}}}
```

Meilleur typage des records

$F = \{a : (x \Rightarrow x + 1), ..\}$
$F(\{a : 0, b : true\}) \rightsquigarrow \{a : 1, b : true\}$

Avant :

$F(\{a : int, b : bool\}) : \{a : int, ..\}$

Après :

$F(\{a : int, b : bool\}) : \{a : int, b : bool\}$

Type inference

Environnements

Γ : **Vars** $\to$ **Types**, variables captured by patterns
Δ : **FVars** $\to$ **Filters**, recursion variables for filters (let rec X = ...)
M : Dom(Δ) $\times$ **Types** $\to$ **TVars**, memoisation for filter applications

$$\frac{\Gamma;\Delta;M \vdash_{\text{fil}} f_1(t_1) : s_1 \quad \Gamma;\Delta;M \vdash_{\text{fil}} f_2(t_2) : s_2}{\Gamma;\Delta;M \vdash_{\text{fil}} f_1|f_2\ (t) : \bigvee_{\{\ i \mid t_i \neq \texttt{empty}\ \}} s_i} \quad \begin{array}{l} t \leq \lfloor f_1 \rfloor \mid \lfloor f_2 \rfloor \\ t_1 = t\&\lfloor f_1 \rfloor \\ t_2 = t\&\neg\lfloor f_1 \rfloor \end{array}$$

- $\lfloor f \rfloor$ approximates the set of values v such that $f(v)$ does not fail.
- $v \in \lfloor f \rfloor$ is necessary, not sufficient.

Sous-typage sémantique (2)

Interprétation ensembliste

- $[\![\mathrm{t}]\!] = \{v \in \text{Valeurs} \mid v : t\}$
- $s \leq t \iff [\![\mathrm{s}]\!] \subseteq [\![\mathrm{t}]\!]$
- Vide et donc sous-typage décidable
  $(s \leq t \iff s\&\neg t = \texttt{empty})$
- Expressions régulières arbitraires : `[char+ (int|bool)?]`