Algèbres de collections

E.Coquery

emmanuel.coquery@univ-lyon1.fr

http://emmanuel.coquery.pages.univ-lyon1.fr
→ Enseignement → MIF04 : Gestion de Données pour le Web

Collections

"Paquets" de valeurs :

- Avec ou sans doublons
- Triés ou non
- Contenant des valeurs homogènes

Des opérations sur les collections communes à de nombreux environnements

Lesquelles ?

- Transformer
- Filtrer, extraire
- Combiner
- Agréger
- Trier

Où ?

- Dans les bases de données (SQL, NoSQL)
- Dans les langages de programmation (Javascript, Python, Java, OCaml, etc)

Algèbre

Dans le cadre du cours :

- Ensemble : collections ordonnées de documents JSON / valeurs
  - Le nombre d'occurrences compte
  - L'ordre d'apparition des éléments compte
- Opérations :
  - Map_f
  - $Filter_f$
  - $Join_f$
  - $Agg_{f,g,a}$, $FlatMap_f$
  - $Sort_f$

Mini-langage pour exprimer les calculs

$$
\begin{array}{rcl}
d & := & 1 \mid 2 \mid \ldots \\
 & \mid & 1.0 \mid 5.7 \mid \ldots \\
 & \mid & \texttt{"truc"} \mid \ldots \\
 & \mid & d(d) \mid d_1\ op\ d_2 \\
 & \mid & \lambda x.d \\
 & \mid & \{a_1 : d_1, \ldots, a_n : d_n\} \\
 & \mid & d.a \\
 & \mid & [] \mid [d] \\
op & := & \texttt{+} \mid \texttt{*} \mid \texttt{=} \mid \texttt{++} \mid \ldots
\end{array}
$$

Rmq : $d_1\ op\ d_2$ est une écriture pour $op(d_1)(d_2)$

Types pour les données ($\approx$ JSON Schema)

- de base : *int*, *float*, *string*, *date*, ...
- records : $< field_1 : \tau_1, \ldots, field_k : \tau_k >$
  - Pas d'ordre entre les champs
- collections/tableaux/listes : $arr(\tau)$
- fonctions : $\tau_1 \rightarrow \tau_2$

Types des constantes

$$(\text{Const})\ \frac{}{\Gamma \vdash c : \tau}$$

en prenant τ et c comme suit :

- type *int* : 1 , 2 , ...
- type *float* : 1.0 , 0.3 , 10.42 , ...
- type *string* : "truc" , ...

Typage des fonctions

$$\text{(App)}\ \frac{\Gamma \vdash f : \tau \to \tau' \qquad \Gamma \vdash d : \tau}{\Gamma \vdash f(d) : \tau'}$$

$$\text{(Lambda)}\ \frac{\Gamma[x : \tau] \vdash d : \tau'}{\Gamma \vdash \lambda x.d : \tau \to \tau'}$$

$$\text{(Var)}\ \frac{}{\Gamma \vdash x : \tau}\ \text{si}\ x : \tau \in \Gamma$$

Typage des records, des listes

$$\text{(Field)}\ \frac{\Gamma \vdash d :< a : \tau >}{\Gamma \vdash d.a : \tau}$$

$$\text{(Record)}\ \frac{\Gamma \vdash d_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash d_n : \tau_n}{\Gamma \vdash \{a_1 : d_1, \ldots, a_n : d_n\} :< a_1 : \tau_1, \ldots, a_n : d_n >}$$

$$\text{(Singleton)}\ \frac{\Gamma \vdash d : \tau}{\Gamma \vdash [d] : arr(\tau)}$$

$$\text{(Empty)}\ \frac{}{\Gamma \vdash [] : arr(\tau)}$$

Sous-typage

$$\text{(Sous-typage)}\ \frac{\Gamma \vdash d : \tau \qquad \tau \preccurlyeq \tau'}{\Gamma \vdash d : \tau'}$$

Sous-types

$$\text{(Refl)}\ \frac{}{\tau \preceq \tau} \qquad \text{(Trans)}\ \frac{\tau \preceq \tau' \qquad \tau' \preceq \tau''}{\tau \preceq \tau''}$$

$$\frac{}{< a_1 : \tau_1, \ldots, a_k : \tau_k, a_{k+1} : \tau_{k+1} > \preceq < a_1 : \tau_1, \ldots, a_k : \tau_k >}\ \text{(AddField)}$$

$$\text{(SubField)}\ \frac{\tau_k \preceq \tau'_k}{< a_1 : \tau_1, \ldots, a_k : \tau_k > \preceq < a_1 : \tau_1, \ldots, a_k : \tau'_k >}$$

$$\text{(SubList)}\ \frac{\tau \preceq \tau'}{arr(\tau) \preceq arr(\tau')}$$

$$\text{(SubFunc)}\ \frac{\tau_1 \succeq \tau'_1 \qquad \tau_2 \preceq \tau'_2}{\tau_1 \to \tau_2 \preceq \tau'_1 \to \tau'_2}$$

Fonctions pures

Pure : Résultat de la fonction ne dépend que de ses arguments, pas du contexte.

Dans la suite du cours, on supposera que toutes les fonctions qui paramètrent des opérateurs sont pures.

Collections

Contenu :

- records, qui ne contiennent que des données
  - pas de fonction
  - pas de variable
- tous les éléments ont le même type
  (mais sous-typage autorisé)

Algèbre

Opérateurs paramétrés par :

- des fonctions pures
- des noms de champ

$$Map_f,\ Filter_f,\ Join_f,\ Agg_{f,a},\ FlatMap_f,\ Sort_f$$

Map_f

Transforme chaque élément de la collection via f

- $f : \tau \to \tau'$
- $Map_f : arr(\tau) \to arr(\tau')$

- SQL : `SELECT`
- Mongo agg : `$project` ou `$replaceWith`

  Reformuler f avec des expressions définissant des attributs

- Javascript : `Array.prototype.map`
- Python : `map(f, ...)`
- OCaml : `List.map`
- Java : `Stream.map`

$Filter_f$

Conserve uniquement certains éléments, choisis par f

- $f : \tau \to bool$
- $Filter_f : arr(\tau) \to arr(\tau)$

- SQL : `WHERE`
  Reformuler f avec des conditions sur les attributs
- Mongo agg : `$match`
  Reformuler f sous forme de conditions (*query*)

- Javascript : `Array.prototype.filter`
- Python : `filter(f, ...)`
- OCaml : `List.filter`
- Java : `Stream.filter`

$Join_f$

Combine les éléments de deux collections.
e_1 est combiné avec e_2 si $f(e_1)(e_2) = true$

- $f : \tau_1 \to \tau_2 \to bool$
- $Join_f : arr(\tau_1) \to arr(\tau_2) \to arr(< left : \tau_1, right : \tau_2 >)$

- SQL : `JOIN`
  Reformuler f sous forme de conditions dans le `ON`
- Mongo agg : `$lookup`
  Reformuler f sous forme de conditions (*query*), et faire suivre d'un `$unwind`

- Javascript, Python, OCaml, Java : pas de codage direct
- `coll1.map(e1 => coll2.filter(e2 => f(e1,e2)).map(e2 => {"left": e1, "right": e2})).flat()`

$Agg_{f,g,a}$

Regroupe les éléments de la collection selon les valeur de a. Créée un record pour chaque groupe ayant :

- la valeur de a du groupe dans le champ *key*
- la valeur de $f \circ Map_g$ appliquée au groupe dans le champ *value*

- $f : arr(\tau') \to \tau''$, $g : \tau \to \tau'$
- $Agg_{f,g,a} : arr(\tau) \to < key : \tau_a, value : \tau'' >$, avec $\tau \preccurlyeq < a : \tau_a >$

$Agg_{f,g,a}$ dans les langages concrets

- SQL : `GROUP BY` + fonctions d'aggrégations
- MongoDB : `$group`
- Javascript : à recoder
- Python : `iterttools.groupby`
- OCaml : à recoder
- Java : `Collectors.groupingBy`

$ReduceByKey_{f,a_k,a_v}$: cas particulier de $Agg_{f,g,a}$

Version où l'aggrégation se fait élément par élément, deux à deux en combinant avec des résultats intermédiaires *de même type*.

- $f : \tau_v \to \tau_v \to \tau_v$
- $ReduceByKey_{f,a_k,a_v} : arr(\tau) \to < key : \tau_k, value : \tau_v >$, avec $\tau \preccurlyeq < a_k : \tau_k, a_v : \tau_v >$

$FlatMap_f$

Produit pour chaque élément de la collection initiale des valeurs.
Le résultat est la collection de toutes les valeurs produites.

- $f : \tau \rightarrow arr(\tau')$
- $FlatMap_f : arr(\tau) \rightarrow arr(\tau')$

- SQL : Possible avec certaines fonctions particulières (*e.g.* `unnest` en PostgreSQL)
- MongoDB : `$unwind`

- Javascript : `Array.prototype.flatMap()`
- Python : `itertools.chain.from_iterab`
- OCaml : `List.flatten` combiné avec `List.map`
- Java : `Stream.flatMap`

$Sort_f$

Trie la collection selon la fonction de comparaison f

- $f : \tau \to \tau \to bool$
- $Sort_f : arr(\tau) \to arr(\tau)$

- SQL : `ORDER BY`
- MongoDB : `$sort`

- Javascript : `Array.prototype.sort`
- Python : `sorted`
- OCaml : `List.sort`
- Java : `Stream.sorted`

Union

Assemble des collections

- *Union* : $arr(\tau) \to arr(\tau) \to arr(\tau)$

- SQL : `UNION`
- MongoDB : `$unionWith`

- Javascript : `Array.prototype.concat`
- Python : `itertools.chain`
- OCaml : `List.append`
- Java : `Stream.concat`

Diff

Différence entre collections

- $Diff : arr(\tau) \rightarrow arr(\tau) \rightarrow arr(\tau)$

- SQL : `MINUS`, `NOT IN`, `NOT EXISTS`
- MongoDB : parfois recodable
- Javascript, Python OCaml, Java : recoder avec *Filter* et une fonction de test d'appartenance

Exemples

Exemples avec MongoDB aggregation pipeline

Données à la demande : implémentation basée sur les itérateurs

- Itérateur : objet/fonction/methode fournissant les éléments un par un
- Approche naturelle pour traiter des collection lues depuis des fichiers
- Plus compliqué pour les tris, jointures, calculs de groupes :
  - on perd l'aspect flux ;
  - peut nécessiter la *matérialisation* d'une collection

Digression : Itérateurs en Python

Principle

- Objet avec état utilisé pour itérer sur une collection (possiblement virtuelle)
- `next()` méthode qui renvoie (ou *yields*) le prochain élément
  - throws `StopIteration` lorsqu'il n'y a plus d'éléments.

Digression : Générateurs en Python

- “fonction” spéciale qui créée un itérateur
- `yield` statement :
  - chaque utilisation de `yield` fourni la valeur qui sera renvoyée par le prochain appel à `next()`

Digression : Exemple de générateur

```
def foo():
    print("begin")
    for i in range(3):
        print("before yield", i)
        yield i
        print("after yield", i)
    print("end")

for i in foo():
    print("obtained", i)
```

```
begin
before yield 0
obtained 0
after yield 0
before yield 1
obtained 1
after yield 1
before yield 2
obtained 2
after yield 2
end
```

Générateur pour Map_f

```
def op_map(f, coll):
    for elt in coll:
        yield f(elt)
```

Générateur pour $Filter_f$

```
def op_filter(f, coll):
    for elt in coll:
        if f(elt):
            yield elt
```

Générateur pour $Join_f$

```
def op_join(f, coll1, coll2):
    # Nested loops
    for elt1 in coll1:
        for elt2 in coll2:
            if f(elt1, elt2):
                yield {"left": elt1, "right": elt2}
```

Générateur pour $Agg_{f,g,a}$

```
def op_agg(f, g, a, coll):
    groups = dict()
    for elt in coll:
        if elt[a] not in groups:
            groups[elt[a]] = []
        groups[elt[a]].append(g(elt))
    for k in groups:
        yield {"key": k, "value": f(groups[k])}
```

Générateur pour $Agg_{f,g,a}$, utilisant le tri

```
def op_agg(f, g, a, coll):
    coll = sorted(coll, a)
    k = None
    grp = []
    for elt in coll:
        if k != elt[a] and k is not None:
            yield {"key": k, "value": f(grp)}
            k = elt[a]
            grp = []
        elif k is None:
            k = elt[a]
        grp.append(g(elt))
    if k is not None:
        yield {"key": k, "value": f(grp)}
```

Générateur pour $FlatMap_f$

```
def op_flatmap(f, coll):
    for elt in coll:
        for elt2 in f(elt):
            yield elt2
```

Et en distribué ?

- Collection répartie sur des serveurs $S_1, \ldots, S_k$
- Résultat d'un calcul : union des résultats produits par chaque serveur
- Rien de particulier à faire pour Map_f, $Filter_f$, $FlatMap_f$

$Sort_f$ en distribué

- Trier en local sur chaque S_i
- Un des serveurs est élu pour fournir le résultat (arbitrairement S_1)
- S_1 demande à tous les serveurs leur premier élément
- On itère ensuite, jusqu'à ce qu'il n'y ait plus de valeur :
  - déterminer i comme le numéro du serveur ayant produit la plus petite valeur
  - `yield` la valeur
  - mettre à jour la valeur dur serveur i

$Agg_{f,g,a}$ en distribué

- Chaque valeur de a se voit attribuer un serveur
  - e.g. via un hash entre 1 et k
- Chaque serveur redistribut ses éléments en les envoyant vers le serveur en fonction de la valeur de a
- On applique ensuite l'algorithme local

$Join_f$ en distribué

$$\text{Si } f = \lambda e_1.\lambda e_2.e_1[a] = e_2[a]$$

- Chaque valeur de a se voit attribuer un serveur
  - e.g. via un hash entre 1 et k
- Chaque serveur redistribut ses éléments en les envoyant vers le serveur en fonction de la valeur de a
- Calcule pour chaque valeur de a le sous-produit cartésien des deux sous-collections correspondantes