MIF04 – Gestion de données pour le Web
TD – Algèbres de collections

Exercice 1 : Typage En supposant les types suivants pour les *builtins* :

— $(>) : int \to int \to bool$
— $(+) : int \to int \to int$
— $sum : arr(int) \to int$
Donner le type des expressions suivantes :

1. $\lambda r.Map_{\lambda x.(\{A{:}x.A, D{:}x.B+x.C\})}(Filter_{\lambda x.(x.A>x.C)}(r))$
2. $\lambda r.Agg_{\mathtt{sum},\lambda x.(x.A),B}(Filter_{\lambda x(x.C>0)}(r))$
3. $\lambda r.Map_{\lambda x.(\{D{:}x.left.A, E{:}x.right.C\})}(Join_{\lambda y.\lambda z.(y.B=z.B)}(r)(r))$

Exercice 2 : Sous-typage

1. Expliquer, en vous appuyant éventuellement sur des exemples, le fonctionnement des règles de sous-types des records (AddField) et (SubField).
2. Expliquer en quoi la règle de sous-type (AddField) est compatible avec le système de types (i.e. on n'accède pas à des champs de records qui n'existent pas).

Exercice 3 : Algèbre en SQL On suppose une relation $R(A, B, C)$ représentée par une collection R de type $< A : int, B : int, C : int >$. Pour chacune des requêtes d'algèbre de collection suivantes, donner un équivalent en SQL.

1. $Map_{\lambda x.(\{A{:}x.A, D{:}x.B+x.C\})}(Filter_{\lambda x.(x.A>x.C)}(R))$
2. $Agg_{\mathtt{sum},\lambda x.(x.A),B}(Filter_{\lambda x(x.C>0)}(R))$
3. $Map_{\lambda x.(\{D{:}x.left.A, E{:}x.right.C\})}(Join_{\lambda y.\lambda z.(y.B=z.B)}(R)(R))$

Exercice 4 : SQL en Algèbre On suppose une relation $R(A, B, C)$ représentée par une collection R de type $< A : int, B : int, C : int >$. Pour chacune des requêtes SQL suivantes, donner un équivalent en algèbre de collections, en Javascript puis en MongoDB aggregation pipeline.

1.

```
SELECT *
FROM R
WHERE R.A=R.B+R.C
```

2.

```
SELECT SUM(A) as D, B, C
FROM R
GROUP BY B, C
ORDER BY B, C
```

3.

```
SELECT R.A, R.C, R2.D
FROM R
JOIN (SELECT count(R.A) as D, C
      FROM R
      GROUP BY C) R2
  ON R.C = R2.C
```

4.

```
SELECT R1.C, SUM(R1.A)+COUNT(R2.B) as D
FROM R R1 JOIN R R2 ON R1.C = R2.C
GROUP BY R1.C
HAVING R1.C < COUNT(R1.B)
```

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

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

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

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

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

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"` , ...

Sous-typage

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

$$\text{(Refl)}\ \frac{}{\tau \preccurlyeq \tau}$$

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

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

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

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

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

FIGURE 1 – Typage

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

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

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

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

$FlatMap_f : arr(\tau) \to arr(\tau')$
avec $f : \tau \to arr(\tau')$

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

FIGURE 2 – Types des opérateurs de l'algèbre

En MongoDB, on peut appeler le framework d'aggrégation via `db.collection.aggregate(...)` Voici quelques *stages* utilisables dans le pipeline :

$project spec : ensemble de spécifications de champs. Permet d'ajouter, supprimer ou changer la valeurs de certains champs.

$match spec : conditions sur les champs (c.f. query Mongo). Filtre les éléments du pipeline à la façon d'une requête MongoDB classique.

$lookup spec :

```
{
    from: <collection to join>,
    localField: <field from the input documents>,
    foreignField: <field from the documents of the "from" collection>,
    as: <output array field>
}
```

Effectue une jointure avec la collection spécifiée, mais en combinant chaque docuemnt de la collection courante avec le tableau des documents correspondants.

$unwind spec : chemin de champs avec $ au début. Produit un document pour chaque valeur du tableau contenu dans le champ spécifié en recopiant le reste du document.

$group spec :

```
{
    _id: <expression>, // Group By Expression
    <field1>: { <accumulator1> : <expression1> },
    ...
}
```

Regroupe les documents par valeur décrite dans `_id` et calcule pour chaque champ la valeur aggrégée décrite par l'accumulateur auquel on passe les valeurs calculées par l'expression.

$sort spec : ensemble de champs avec le ordre de tri (1 ou -1)

Réf : `https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/`

FIGURE 3 – Aggrégation pipeline en MongoDB

Corrections

Solution de l'exercice 1 On donne quelques dérivations car elles permettent d'expliciter les types, mais il faut être capable de trouver ces types directement. On pose $\Gamma = (>) : int \to int \to bool\,,\ (+) : int \to int \to int\,,\ sum : arr(int) \to int$.

Rappel : $a > b$ est juste une écriture pour $((>)(a))(b)$

1. On remarque que $< A : int, B : int, C : int > \preccurlyeq < A : int >$ via la dérivation suivante :

$$\dfrac{\dfrac{}{< A : int, B : int, C : int > \preccurlyeq < A : int, B : int >}\,(\text{AddField}) \quad \dfrac{}{< A : int, B : int > \preccurlyeq < A :}\,(\text{AddField})}{< A : int, B : int, C : int > \preccurlyeq < A : int >}\,(\text{Trans})$$

De même, $< A : int, B : int, C : int > \preccurlyeq < C : int >$

On pose $\Gamma' = \Gamma, x :< A : int, B : int, C : int >$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash (>) : int \to int \to bool}\,(\text{Var}) \quad \overset{(D_1)}{\Gamma' \vdash x.A : int}}{\Gamma' \vdash (>)(x.A) : int \to bool}\,(\text{App}) \quad \overset{(D_2)}{\Gamma' \vdash x.C : int}}{\Gamma' \vdash ((>)(x.A))(x.C) : bool}\,(\text{App})}{\Gamma \vdash \lambda x.(>)(x.A)(x.C) : < A : int, B : int, C : int > \to bool}\,(\text{Lambda})$$

(D_1) :

$$\dfrac{< A : int, B : int, C : int > \preccurlyeq < A : int > \qquad \dfrac{\dfrac{}{\Gamma' \vdash x :< A : int, B : int, C : int >}\,(\text{Var})}{\Gamma' \vdash x :< A : int >}\,(\text{Field})}{\Gamma' \vdash x.A : int}\,(\text{Sous-typage})$$

(D_2) :

$$\dfrac{< A : int, B : int, C : int > \preccurlyeq < A : int > \qquad \dfrac{\dfrac{}{\Gamma' \vdash x :< A : int, B : int, C : int >}\,(\text{Var})}{\Gamma' \vdash x :< C : int >}\,(\text{Field})}{\Gamma' \vdash x.C : int}\,(\text{Sous-typage})$$

On a donc $\Gamma \vdash \lambda x.(x.A > x.C) : < A : int, B : int, C : int > \to bool$.

Remarque : on peut aussi dériver que $\Gamma \vdash \lambda x.(x.A > x.C) : < A : int, C : int > \to bool$ (sans le champ B). C'est même plus naturel. Mais en regardant la suite du calcul, on voit qu'on aura besoin du champ B.

En procédant de manière similaire, on déduit que

$$\Gamma \vdash \lambda x.(\{A : x.A, D : x.B + x.C\}) :< A : int, B : int, C : int > \to < A : int, D : int >$$

En utilisant les informations de la figure 2, on peut déduire les types suivants :

$$\underbrace{\lambda r.\, \underbrace{Map_{\lambda x.(\{A:x.A, D:x.B+x.C\})}(\underbrace{Filter_{\lambda x.(x.A>x.C)}(r)}_{arr(<A:int,B:int,C:int>)})}_{arr(<A:int,D:int>)}}_{arr(<A:int,B:int,C:int>) \to arr(<A:int,D:int>)}$$

2. On a les types suivants pour les fonction qui apparaissent ci-dessous :
   — $\Gamma \vdash \texttt{sum} : arr(int) \to int$
   — $\Gamma \vdash \lambda x(x.C > 0) :< C : int >\to bool$ et donc par sous-typage $\Gamma \vdash \lambda x(x.C > 0) :< A : int, B : \tau_B, C : int >\to bool$
   — $\Gamma \vdash \lambda x.(x.A) :< A : int >\to int$ et donc par sous-typage $\Gamma \vdash \lambda x(x.A) :< A : int, B : \tau_B, C : int >\to int$

   On peut donc déduire les types suivants

$$\underbrace{\lambda r.\, \underbrace{Agg_{\texttt{sum},\lambda x.(x.A),B}(\ \underbrace{Filter_{\lambda x(x.C>0)}(r)}_{arr(<A:int,B:\tau_B,C:int>)}\ )}_{arr(<key:\tau_B,value:int>)}}_{arr(<A:int,B:\tau_B,C:int>)\to arr(<key:\tau_B,value:int>)}$$

3. Pour faire plus court dans les notations, on pose $\tau_r =< A : \tau_A, B : \tau_B, C : \tau_C >$. On a les types suivants pour les fonction qui apparaissent ci-dessous :
   — $\Gamma \vdash \lambda y.\lambda z.(y.B = z.B) :< B : \tau_B >\to< B : \tau_B >\to bool$, donc, par sous-typage, $\Gamma \vdash \lambda y.\lambda z.(y.B = z.B) : \tau_r \to \tau_r \to bool$
   — $\Gamma \vdash \lambda x.\{D : x.left.A, E : x.right.C\} :< left :< A : \tau_A >, right :< C : \tau_C >>\to< D : \tau_A, E : \tau_C >$, donc, par sous-typage, $\Gamma \vdash \lambda x.\{D : x.left.A, E : x.right.C\} :, < left : \tau_r, right : \tau_r >\to< D : \tau_A, E : \tau_C >$

$$\underbrace{\lambda r.\, \underbrace{Map_{\lambda x.(\{D:x.left.A,E:x.right.C\})}(\underbrace{Join_{\lambda y.\lambda z.(y.B=z.B)}(r)(r)}_{arr(<left:\tau_r,right:\tau_r>)})}_{arr(<D:\tau_A,E:\tau_C>)}}_{arr(\tau_r)\to arr(<D:\tau_A,E:\tau_C>)}$$

Solution de l'exercice 2

1. La règle (AddField) dit qu'un type record ayant un champ de plus qu'un autre (les autres champs étant identiques) est plus petit dans l'ordre de sous-typage. Si on combine cela avec la règle de sous-typage, cela veut dire que, si cela arrange pour typer, on peut oublier un champ dans un record.
   Par exemple quand on dit que $< A : int, B : string > \preccurlyeq < B : string >$, on oublie le champ A.
   La règle (SubField) indique que si pour un champ dans un premier type record τ on a un type plus petit que pour le même champ dans un type τ' (les types des autres champs étant identiques par ailleurs), alors le type τ est plus petit dans l'ordre des types que le type τ'.
   Si on combine cela avec la règle de sous-typage et avec la règle (AddField), cela veut dire que, si cela arrange pour typer, on peut oublier un champ dans un record imbriqué dans le record considéré.
   Par exemple, on peut oublier le champ A dans le record placé dans le champ C : $< C :< A : int, B : string >, D : int > \preccurlyeq < C :< B : string >, D : int >$.
2. La règle (AddField) dit que l'on peut oublier certains champs *du point de vue des types*. C'est compatible avec le système de type car le language ne permet que de lire des records ou de les fabriquer, mais pas de les stocker dans un endroit. Cela ne serait pas le cas dans

un language impératif, comme par exemple le C. En C, si on a une variable qui contient un `struct` avec deux champs `a` et `b` on ne peut pas y ranger un `struct` avec des champs `a`, `b` et `c`.

Solution de l'exercice 3

1. $Map_{\lambda x.(\{A:x.A,D:x.B+x.C\})}(Filter_{\lambda x.(x.A>x.C)}(R))$

```
SELECT A, B+C as D
FROM R
WHERE A > C
```

2. $Agg_{\mathtt{sum},\lambda x.(x.A),B}(Filter_{\lambda x(x.C>0)}(R))$

```
SELECT B as key, SUM(A) AS value
FROM R
WHERE C > 0
GROUP BY B
```

3. $Map_{\lambda x.(\{D:x.left.A,E:x.right.C\})}(Join_{\lambda y.\lambda z.(y.B=z.B)}(R)(R))$

```
SELECT R1.A as D, R2.C as E
FROM R R1
JOIN R R2 ON R1.B = R2.B
```

Solution de l'exercice 4 Pour la partie MongoDB, on peut se donner une collection de départ pour tester :

```
db.r.insertMany([
    { "A" : 1, "B" : 2, "C" : 3 },
    { "A" : 5, "B" : 2, "C" : 5 },
    { "A" : 5, "B" : 2, "C" : 3 },
    { "A" : 8, "B" : 5, "C" : 3 },
    { "A" : 8, "B" : 5, "C" : 5 },
    { "A" : 1, "B" : 1, "C" : 1 },
    { "A" : 1, "B" : 2, "C" : 1 },
    { "A" : 1, "B" : 1, "C" : 2 },
    { "A" : 1, "B" : 2, "C" : 2 }
])
```

1.

```
SELECT *
FROM R
WHERE R.A=R.B+R.C
```

Algèbre

$$Filter_{\lambda x.(x.A=x.B+x.C)}(R)$$

MongoDB

```
db.r.aggregate([
    { $match: { $expr: { $eq: [ "$A",
                                { $add: [ "$B", "$C" ] } ] }}}
])
```

Javascript

```
r.filter(x => (x.A == x.B+x.C))
```

2.

```
SELECT SUM(A) as D, B, C
FROM R
GROUP BY B, C
ORDER BY B, C
```

Algèbre

$$Sort_{\lambda x.\lambda y.(x.B<y.B \text{ or } (x.B=y.B \text{ and } x.C<y.C))}($$
$$Map_{\lambda x.\{D:x.value,B:x.key.B,C:x.key.C\}}($$
$$Agg_{sum,\lambda x.(x.A),cle}($$
$$Map_{\lambda x.\{A:x.A,cle:\{B:x.B,C:x.C\}\}}(R))))$$

MongoDB

```
db.r.aggregate([
    { $group: { _id: { B: "$B", C: "$C" },
                D: { $sum: "$A" } }},
    { $project: { _id: 0,
                  D: 1,
                  B: "$_id.B",
                  C: "$_id.C" } },
    { $sort: { B: 1, C:1 } },
])
```

Javascript

```
var groupBy = a => c => {
    let m = {}
    c.forEach(elt => {
        let key = JSON.stringify(elt[a]);
        if (m[key] === undefined) {
            m[key]=[]
        }
        m[key].push(elt);
    })
    var result = [];
    for (k in m) {
        result.push({"key":JSON.parse(k), "values": m[k]});
    }
    return result;
};

groupBy("cle")(r.map(x => ({ cle: { "B": x.B, "C": x. C }, A: x.A})))
    .
```

```
map(x => ({ "B": x.key.B, "C": x.key.C, D: x.values.reduce((x,y)
    => x+y.A, 0)})).
sort((x,y) => x.B < y.B ? -1 :
              (x.B > y.B ? 1 :
              (x.C < y.C ? -1:
              (x.C > y.C ? 1 : 0))))
```

3.
```
SELECT R.A, R.C, R2.D
FROM R
JOIN (SELECT count(R.A) as D, C
      FROM R
      GROUP BY C) R2
  ON R.C = R2.C
```

Algèbre

$$Map_{\lambda x.\{A:x.left.A,C:x.left.C,D:x.right.value\}}(Join_{\lambda x.\lambda y.(x.C=y.key)}(R)(Agg_{length,\lambda x.(x.A),C}(R)))$$

MongoDB

```
db.r.aggregate([
    { $group: { _id: "$C",
                D: { $sum: 1 } } },
    { $lookup: { from: "r",
                 localField: "_id",
                 foreignField: "C",
                 as: "R" } },
    { $unwind: "$R" },
    { $project: {
        _id: 0,
        A: "$R.A",
        C: "$R.C",
        D:1,
    }},
])
```

Javascript En reprenant la définition de groupBy précédente :

```
var join = (f) => (r1,r2) => {
    let res = [];
    for (let e1 of r1) {
        for (let e2 of r2) {
            if (f(e1,e2)) {
                res.push({"left": e1, "right": e2})
            }
        }
    }
    return res;
};

join((x,y) => (x.C==y.C))(r, groupBy("C")(r).map(x =>({"C":x.key, "D"
    :x.values.length})))).
    map(x => ({"A": x.left.A, "C": x.left.C, "D": x.right.D}))
```

4.
```sql
SELECT R1.C, SUM(R1.A)+COUNT(R2.B) as D
FROM R R1 JOIN R R2 ON R1.C = R2.C
GROUP BY R1.C
HAVING R1.C < COUNT(R1.B)
```

Algèbre On pose :

$$f_{aggr} = \lambda x.\{ \quad SU : sum(Map_{\lambda y.(y.A1)}), \\ CO1 : length(Map_{\lambda y.(y.B1)}), \\ CO2 : length(Map_{\lambda y.(y.B2)})\}$$

$$Map_{\lambda x.\{C:x.key, D:x.value.SU+x.value.CO2\}}(\\ Filter_{\lambda x.(x.key<x.value.CO1)}(\\ Agg_{f_{aggr},\lambda x.x,C1}(\\ Map_{\lambda x.\{A1:x.left.A, B1:x.left.B, B2:x.right.B, C1:x.left.C\}}(\\ Join_{\lambda x.\lambda y.(x.C=y.C)}(R)(R)))))$$

MongoDB

```
db.r.aggregate([
    { $lookup: {
        from: "r",
        localField: "C",
        foreignField: "C",
        as: "R2"
    }},
    { $unwind: "$R2" },
    { $group: { _id: "$C",
                SU: { $sum: "$A" },
                CO1: { $sum: 1 },
                CO2: { $sum: 1 } }},
    { $match: { $expr: { $lt: [ "$_id", "$CO1" ] } }},
    { $project: {
        C: "$_id",
        _id: 0,
        D: { $add: [ "$SU", "$CO2" ] }
    }}
])
```

Javascript En réutilisant groupBy et `join` définis précédement :

```
groupBy("C")(join((x,y)=>(x.C==y.C))(r,r).
                map(x => ({C: x.left.C, A: x.left.A}))).
    map(x => ({C: x.key,
               SU: x.values.map(y => y.A).reduce((e1,e2)=>e1+e2),
               CO1: x.values.length,
               CO2: x.values.length })).
    filter(x => x.C < x.CO1).
    map(x => ({C: x.C, D: x.SU+x.CO2}))
```