SQL à l'épreuve de Coq Une sémantique formelle pour SQL

Véronique Benzaken, Evelyne Contejean

SQL à l'épreuve de Coq
Une sémantique formelle pour SQL *

Véronique Benzaken[1] and Évelyne Contejean[2]

[1] Université de Paris Sud - Université Paris Saclay
veronique.benzaken@u-psud.fr
[2] CNRS - Université de Paris Sud - Université Paris Saclay
evelyne.contejean@lri.fr

Résumé

Nous proposons une sémantique formelle, exécutable, mécanisée en Coq pour un fragment réaliste du langage SQL, le standard en terme de bases de données relationnelles. Ce fragment prend en compte des requêtes de la forme `select [distinct] from where group by having` en présence de *valeurs nulles, fonctions, agrégats, quantificateurs* ainsi que de requêtes *imbriquées* potentiellement *corrélées*. Nous relions ce fragment à l'algèbre relationnelle, équivalent pour SQL du λ-calcul pour les langages fonctionnels. Nous obtenons ainsi le premier résultat d'équivalence de ce fragment et de l'algèbre, formellement prouvé. Enfin, grâce au mécanisme d'extraction, nous fournissons un analyseur sémantique du langage SQL certifié en Coq, élément central pour l'obtention d'un compilateur SQL vérifié.

1 Introduction

Définir la sémantique formelle d'un langage est une tâche délicate mais cruciale pour permettre de raisonner rigoureusement sur le comportement des programmes et pour vérifier statiquement la correction des optimisations [13]. La formalisation des langages les plus répandus peut s'avérer ardue puisqu'ils sont fréquemment spécifiés par des documents rédigés en langue naturelle, et de ce fait ambigus. Même lorsqu'il existe une spécification formelle, elle est incomplète, ne couvre qu'un fragment limité du langage et sa correction n'a le plus souvent été vérifiée que par un humain faillible. Dans tous les cas, peu de garanties indiscutables attestent que la sémantique proposée rend fidèlement compte de la sémantique réelle et que les optimisations effectuées sont vraiment correctes. SQL est le langage standard pour les bases de données relationnelles (SGBDR), à ce titre il est largement utilisé et n'échappe pas à ce travers.

Une approche prometteuse pour obtenir des garanties irréfutables est d'utiliser des assistants à la preuve, comme Coq [19] ou Isabelle [20], pour définir des sémantiques exécutables, mécanisées dont la correction est vérifiable par une machine. Le projet CompCert [15] en est une démonstration éclatante. Notre visée à long terme est de développer un compilateur de SQL, certifié en Coq ; dans cet article nous présentons les aspects liés à la sémantique de SQL.

Spécificités de SQL S'il y a eu quelques tentatives pour définir une sémantique formelle pour SQL, elle ne traitent que de fragments limités du langage, ce qui s'explique, en partie, par l'existence de singularités inhérentes au langage. Il est notoire que le bloc `select from where` possède une sémantique *multi-ensembliste*, tandis les opérateurs du langage $\cup$ (`union`), $\cap$ (`intersect`) et $\setminus$ (`except`) ont une sémantique strictement ensembliste. SQL manipule des *valeurs* nulles (`NULL`) pour représenter de l'information inconnue. Dans le domaine des bases

*Financé par le projet ANR DataCert : ANR-15-CE39-0009.

de données, cet aspect est considéré de la plus haute importance. Les requêtes comportent des *fonctions* (`+,-,...`), et des *agrégats* (`sum, max,...`) très utilisés, en particulier par la clause `group by having`. Enfin, SQL permet l'imbrication de sous-requêtes, *i.e.*, des requêtes mettant en jeu différents blocs `select`, et, de façon plus pernicieuse, l'emploi de *requêtes corrélées*, *i.e.*, des sous-requêtes possédant des variables libres comme dans `select a1 from t1 group by a1 having exists (select a2 from t2 group by a2 having sum(1+0*a1) = 0);` où `a1` est libre dans le `select` le plus interne.

Toute sémantique fidèle se doit de rendre compte, avec précision, de ces quatre points qui pris *séparément* sont purement techniques. Les considérer *ensemble*, comme nous le faisons, constitue un défi dû à la gestion étrange des environnements dans le langage. Ceci explique, sans doute, l'absence à ce jour, d'une sémantique formelle pour SQL.

État de l'art La définition d'une sémantique formelle pour SQL a fait et fait encore l'objet de recherches actives dans la communauté des bases de données. L'algèbre relationnelle [8] étant à SQL ce que le λ-calcul est aux langages fonctionnels, c'est tout naturellement que les premiers travaux [5, 17] ont proposé des traductions de SQL vers l'algèbre, mais d'une version restreinte de SQL, sans valeurs nulles ni fonctions, ni requêtes imbriquées ni multi-ensembles. La première sémantique prenant les valeurs nulles et les multi-ensembles en compte est donnée par [12], mais encore une fois sans la clause `group by having` ni les fonctions ni les agrégats. Or, comme nous le montrerons en Sections 2 et 3, le traitement de ces constructions est particulièrement épineux.

Sur le versant des assistants à la preuve, la première tentative de formalisation, en Agda [18], de l'algèbre relationnelle est proposée par [10, 9] tandis que la première formalisation, complète, du modèle relationnel est proposée par [3] où le modèle de données, l'algèbre, les requêtes "tableaux", la procédure de semi-décision "chase" et les contraintes d'intégrité sont formalisés.

La toute première tentative de vérifier, en Coq, un SGBDR est présentée dans [16]. Cependant, le fragment considéré est une reconstruction de SQL dans laquelle les attributs des relations sont représentés par leur position. Ni la clause `group by having`, ni les quantificateurs dans les formules, les valeurs nulles, les agrégats ou les requêtes imbriquées ne sont traités. Un outil permettant de décider de l'équivalence de requêtes SQL est présenté dans [6]. À cette fin, HottSQL, une sémantique basée sur la notion de K-relation [11] est définie. Elle prend en compte le bloc `select from where` avec agrégats. Comme [16], une reconstruction du langage est utilisée, ce qui évite de se pencher sur les aspects les plus délicats relatifs à la gestion des environnements. Enfin, et surtout, cette sémantique n'est pas *exécutable* : il est impossible de vérifier qu'elle est conforme à celle de SQL. La proposition la plus proche de la nôtre en terme de sémantique mécanisée est présentée dans [2]. Les auteurs modélisent en Coq l'algèbre relationnelle imbriquée (NRA [7]) qui sert *directement* de sémantique à SQL. Ils assignent une sémantique par traduction. Cependant, la syntaxe de NRA s'écartant considérablement de celle de SQL, il n'est pas immédiat de se convaincre que la sémantique assignée reflète fidèlement celle du langage. Nous dissocions l'algèbre et sa sémantique de celle de SQL. C'est en cela que notre approche diffère de celle proposée par [2].

Contributions Cet article présente cinq contributions. Contrairement aux approches présentées dans les articles [16, 6], nous définissons (i) SQL_{Coq} (syntaxe et sémantique), une formalisation en Coq de SQL prenant en compte le bloc complet `select [distinct] from where group by having` en présence de *valeurs* `NULL` *fonctions, agrégats, quantificateurs* et de (sous) requêtes *imbriquées* potentiellement *corrélées*. Ce faisant, grâce au mécanisme d'extraction nous obtenons ainsi (ii) un analyseur sémantique pour SQL certifié en Coq. Nous formalisons en Coq (iii) SQL_{Alg} une algèbre relationnelle multi-ensembliste similaire, quoique l'étendant, à celle présentée dans [8]. Nous définissons, en Coq, les traductions de SQL_{Coq} vers SQL_{Alg} et vice versa et démontrons un théorème de préservation des sémantiques. Ceci permet de (iv) recouvrer

toutes les équivalences algébriques sur lesquelles se fondent les optimisations du compilateur et (v) d'obtenir le premier, à notre connaissance, résultat d'équivalence, formellement prouvé, entre ce fragment de SQL et l'algèbre relationnelle. Le développement est disponible à l'url : http://datacert.lri.fr/sqlcert/.

Organisation La Section 2, présente SQL et les spécificités devant être prises en compte afin de fournir une sémantique fidèle pour un fragment réaliste du langage. Nous définissons en Section 3, la syntaxe et la sémantique mécanisées de SQL_{Coq}. La Section 4 se concentre sur la mécanisation de SQL_{Alg}, la définition des traductions et la preuve du théorème d'adéquation. Nous concluons, tirons des enseignements et donnons nos perspectives en Section 5.

2 SQL : simple ... quoique

SQL est un langage déclaratif dédié à la manipulation de données stockées au sein de bases de données relationnelles. À cet égard il est souvent considéré comme simple et intuitif d'utilisation.

2.1 SQL en bref

SQL opère sur des collections de *n-uplets*. Les n-uplets sont des enregistrements étiquetés dont les champs prennent leurs valeurs dans des domaines atomiques (entiers, chaînes etc.). Dans ce cadre les étiquettes, appelées *attributs*, sont dénotables et sont utilisés dans des expressions construites à partir de fonctions et d'accumulateurs appelés *agrégats*. La structure d'une requête ou bloc SQL est la suivante : `select e from q where c1 group by e' having c2`. Par exemple `select a+2 as a', max(c) as mc from t where b>3 group by a having sum(c)=0` est une requête représentative du langage où `a,b,c` sont des attributs, `sum, max` des agrégats, `+` une fonction et `b>3` une condition.

Un bloc s'évalue ainsi : la requête `q` est évaluée, puis filtrée par la condition `c1`. Ce résultat intermédiaire est "groupé" c'est à dire partitionné grâce aux expressions `e'` qui s'évaluent de manière *homogène* sur les éléments de chaque groupe ; les groupes sont maximaux au sens de l'inclusion. Les groupes sont ensuite filtrés par la condition `c2`, et les expressions `e` sont évaluées pour chaque groupe produisant ainsi le résultat. Pour l'exemple précédent, en supposant que l'évaluation de `t` soit

$$\{(\texttt{a=1;b=1;c=4}), (\texttt{a=1;b=5;c=2}), (\texttt{a=1;b=4;c=-2}), (\texttt{a=3;b=5;c=1}), (\texttt{a=3;b=4;c=2})\},$$

l'étape de filtrage avec `b>3` donne

$$\{(\texttt{a=1;b=5;c=2}), (\texttt{a=1;b=4;c=-2}), (\texttt{a=3;b=5;c=1}), (\texttt{a=3;b=4;c=2})\}.$$

Le groupement selon `a` produit deux groupes

$$\{(\texttt{a=1;b=5;c=2}), (\texttt{a=1;b=4;c=-2})\} \text{ et } \{(\texttt{a=3;b=5;c=1}), (\texttt{a=3;b=4;c=2})\}\}.$$

La seconde étape de filtrage par `sum(c)=0` élimine le dernier groupe pour lequel `sum(c)` vaut `3` et seul le premier groupe est conservé. L'évaluation de `a+2, max(c)` donne le résultat final $\{(\texttt{a'=3;mc=2})\}$.

2.2 Au coeur de SQL

Bien que l'évaluation du bloc soit intuitive, la sémantique de SQL est plus complexe qu'il n'y paraît de prime abord. Elle est décrite par le standard ISO [14] dont il n'est pas aisé de tirer parti car :

- il se compose de milliers de pages où un même aspect se trouve éparpillé en maints endroits du document : il est ainsi complexe de reconstituer toute l'information.
- les fonctionnalités sont délayées (la définition d'un "map" s'étale sur plusieurs pages)
- enfin, quoique abondant, le document présente trop souvent des aspects de manière sous-spécifiée.

En aucun cas le standard n'est exploitable, en l'état, pour offrir une sémantique formelle. Ceci explique, d'ailleurs, pourquoi de nombreux constructeurs en implantent certains traits à leur façon [1]. Nous nous en sommes, bien sûr, servi mais, afin de précisément appréhender la sémantique du langage, nous avons systématiquement confronté notre développement à des systèmes tels PostgreSQL et Oracle™ qui comptent parmi les plus robustes et répandus.

r
a
1
NULL

s
a
NULL

t
a
1
NULL
NULL

t_1

a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1
1	1	1	6	2	1	2	6	3	1	4	6
1	2	1	7	2	2	2	7	3	2	4	7
1	3	1	8	2	3	2	8	3	3	4	8
1	4	1	9	2	4	2	9	3	4	4	9
1	5	1	10	2	5	2	10	3	5	4	10

t_2	
a_2	b_2
7	7
7	7

```
- Q1
select r.a+2 as b from r;
- Q2
select r.a from r where r.a not in (select s.a from s);
- Q3
select r.a from r where not exists
  (select * from s where s.a = r.a);
- Q4
select r.a from r except select s.a from s;
- Q5
select t.a,count(*) as c from t group by t.a;
- Q6
select a1, max(b1) from t1 group by a1;
- Q7(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a2) = k);
- Q8(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a1) = k);
```

```
- Q9(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1) = k);
- Q10(k), k =12
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a1)+sum(1+0*a2) = k);
- Q11(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a1+0*a2) = k);
- Q12(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a1+0*b2) = k);
- Q13(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*b1+0*b2) = k);
- Q14(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*b1+0*a2) = k);
```

FIGURE 1: Un bestiaire de requêtes étranges.

Par la suite nous noterons $[\![q]\!]$ le résultat de l'évaluation de la requête q, () le constructeur de n-uplets, [ ] le constructeur de listes, { } celui d'ensembles et {| |} celui de multi ensembles. La Figure 1 contient un florilège de requêtes illustrant les aspects les plus subtils de SQL.

2.2.1 Valeurs nulles : NULL

La requête `Q1` renvoie {|(`b = 3`); (`b = NULL`)|} illustrant le fait que `NULL` est un élément absorbant pour les fonctions (excepté pour les fonctions Booléennes en PostgreSQL).

La prise en compte des valeurs nulles, dans les formules, se fait au moyen d'une logique trivaluée. Les trois requêtes suivantes, tirées de [12], illustrent que `NULL` n'est ni égal à, ni différent de quelque autre valeur (`NULL` compris) : comparer `NULL` à n'importe quelle expression renvoie toujours `unknown`. `Q2` renvoie un résultat vide. En effet, $[\![$`select s.a from s`$]\!] =$ {|(`s.a=NULL`)|}, donc $[\![$`r.a not in select s.a from s`$]\!]$ (`in` est le prédicat d'appartenance) est évalué à `not unknown`, soit `unknown`, pour tous les n-uplets, en particulier pour (`r.a=1`) et (`r.a=NULL`). Finalement, cette condition sera réduite à `false`, ainsi le résultat de `Q2` est vide. `Q3` renvoie {|(`r.a=1`); (`r.a=NULL`)|}. Considérons `subQ3` : `(select * from s where s.a = r.a)`, son évaluation conduit à un résultat vide pour tous les n-uplets (`r.a=x`), par suite $[\![$`exists (subQ3)`$]\!]$, où `exists` est le prédicat de non vacuité, vaut toujours `false` et $[\![$`not exists (subQ3)`$]\!]$ toujours `true`, donc (`r.a=1`) et (`r.a=NULL`)

sont dans le résultat de `Q3`. La requête `Q4` renvoie $\{(\texttt{r.a=1})\}$, car la différence ensembliste s'appuie sur l'égalité syntaxique standard pour tester l'égalité. Ainsi les n-uplets (`r.a=NULL`) et (`s.a=NULL`) sont égaux. L'évaluation des formules s'opère en logique trivaluée lorsque celles ci sont utilisées comme conditions de filtrage (clause `where` et `having`). Puis SQL plonge de la logique trivaluée à la logique Booléenne usuelle qui est finalement utilisée (`unknown` devient `false`). Ceci sera explicité en Figure 7 et 8 de la Section 3.

Enfin, la requête `Q5` retourne $\{\!|(\texttt{t.a=NULL,c=2});(\texttt{t.a=1,c=1})|\!\}$ et illustre le fait que `NULL`, qui est ni égal ni différent de `NULL` en logique trivaluée est égal à `NULL` dans le contexte du groupement. La sémantique que nous présentons en Section 3.2 rendra compte de tels comportements.

2.2.2 Requêtes corrélées

Intéressons nous à la manière dont SQL gère les environnements et l'évaluation d'expressions en présence d'agrégats et de requêtes imbriquées et corrélées. Pour évaluer des expressions simples (sans agrégats) un environnement contenant l'information relative aux attributs liés et leur valeurs associées suffit. Dans ce cas, par exemple `select a1, b1 from t1;`, un tel environnement consiste en l'unique n-uplet (`a1=`x`,b1=`y) où x et y varient dans les domaines de `a1` et `b1` respectivement.

L'évaluation d' expressions avec agrégats est plus subtile car un agrégat opère sur une liste de valeurs, chacune correspondant à un n-uplet. L'aspect le plus délicat est de comprendre comment une telle liste, que nous appellerons désormais *contexte d'évaluation*, est construite. La Section 10.9 du document ISO [14] (< aggregate functions >, *how to retrieve the rows* – page 545) du standard aurait dû nous éclairer. En vain ! Nous nous sommes donc appuyées sur notre développement SQL_{Coq} qui permet d'exécuter des requêtes (contrairement à [6]) afin de confronter notre sémantique mécanisée à celle de systèmes tels que PostgreSQL et Oracle™. Le recours à une *sémantique mécanisée et exécutable* a été essentiel afin d'examiner tous les sous cas en détail. Cette tâche, quoique chronophage, nous a permis d'obtenir l'ensemble de requêtes pertinentes sur le plan sémantique de la Figure 1. Dans tous les cas nous avons obtenu les même résultats pour les trois systèmes. Détaillons dans ce qui suit le comportement de SQL.

La requête `Q6` retourne $\{\!|(\texttt{a1=1,max=10});(\texttt{a1=2,max=10});(\texttt{a1=3,max=5});(\texttt{a1=4,max=10})|\!\}$. On comprend aisément que `max(b1)` est évalué pour chaque groupe (où `a1` est constant). Par exemple, le groupe T_1 pour lequel `a1=1` contient des n-uplets de la forme (`a1=1,b1=i`), où `i` varie de 1 à 10, ainsi `max(b1)` vaut 10. Le groupe où `a1=3` contient les n-uplets (`a1=3,b1=i`), où `i=1,...,5`, et `max(b1)` vaut 5. Dans ce cas, simple, un groupe correspond naturellement à un contexte d'évaluation. Nous dirons que le groupe a été *scindé* en n-uplets élémentaires.

La situation empire lorsqu'un agrégat est évalué dans une requête imbriquée. Comment SQL construit, dans ce cas, le contexte d'évaluation ? Les dernières requêtes `Q7` à `Q14` et la table `t2` de la Figure 1 ont été conçues afin de clarifier ce point. Notons qu'elles sont toutes corrélées, à l'exception de `Q7` et `Q9`, et ont la forme générale suivante :

```
select a1 from t1 group by a1 having exists (select a2 from t2 group by a2 having e=k);
```

Lorsqu'un aggrégat est présent sous plusieurs niveaux de groupements, dans une requête imbriquée comme c'est le cas pour `e` ci-dessus, plusieurs groupes font partie de *l'environnement*. Pour notre exemple, les groupes homogènes pour `a1` sont :

$$\mathscr{G}_1 = \left\{ \bigcup_{i=1}^{10} \{\!|(\texttt{a1=1;b1=i})|\!\}, \bigcup_{i=1}^{10} \{\!|(\texttt{a1=2;b1=i})|\!\}, \bigcup_{i=1}^{5} \{\!|(\texttt{a1=3;b1=i})|\!\}, \bigcup_{i=6}^{10} \{\!|(\texttt{a1=4;b1=i})|\!\} \right\}$$

alors qu'un seul groupe homogène existe pour `a2`, $T_2 = \{\!|(\texttt{a2=7;b2=7});(\texttt{a2=7;b2=7})|\!\}$. Par conséquent, l'environnement global pour l'expression la plus interne `e`, est constitué de T_2 et d'un élément T_1 de $\mathscr{G}_1$. Nous noterons un tel environnement : $[T_2; T_1]$.

Comment combiner ces groupes pour obtenir le contexte d'évaluation correct ? Quels groupes doivent être scindés ? Considérons `Q7(`k`)`, où la condition de filtrage la plus interne est `sum(1+0*a2)` $= k$. Pour $k \neq 2$, $[\![$`Q7(`k`)`$]\!]$ est vide et pour $k = 2$ le résultat vaut $\bigcup_{i=1}^{i=4} \{|$`(a1=i)`$|\}$. L'expression `sum(1+0*a2)` compte le nombre de n-uplets dans le contexte d'évaluation et ce nombre vaut 2 pour tous les groupes T_1 de $\mathscr{G}_1$. La combinaison de T_1 et T_2 produit un contexte contenant deux (la cardinalité de T_2) n-uplets, quelle que soit la cardinalité de T_1. Nous en tirons la conclusion que lors de l'évaluation de `sum(1+0*a2)` $= k$, T_2 a été scindé tandis que T_1 n'est pas utilisé.

Examinons `Q8(`k`)`, similaire à `Q7(`k`)`, sauf la condition de filtrage qui est `sum(1+0*a1)` $= k$. Pour $k \notin \{5, 10\}$ $[\![$`Q8(`k`)`$]\!]$ est vide alors que $[\![$`Q8(5)`$]\!] = \{|$`(a1=3);(a1=4)`$|\}$ et $[\![$`Q8(10)`$]\!] = \{|$`(a1=1);(a1=2)`$|\}$. L'expression `sum(1+0*a1)` calcule la cardinalité de T_1. Lors de l'évaluation de `sum(1+0*a1)` $= k$, T_1 a été scindé tandis que T_2 n'est pas utilisé.

Constat 1. *Au sein du même environnement, $[T_2; T_1]$, SQL scinde T_1 ou T_2 pour construire son contexte d'évaluation. La façon dont ce contexte est construit est guidée par l'expression à évaluer déterminant quel groupe doit être scindé ou non : T_1 pour* `1+0*a1`*, et T_2 pour* `1+0*a2`*. Un tel choix, en l'espèce, est déterminé par les attributs (*`a1` *ou* `a2`*).*

Il est intéressant, à ce stade, de comprendre le comportement de SQL quand aucun attribut n'est présent dans l'expression sous l'agrégat comme dans `Q9(`k`)`. Quel groupe scinder ? Un tel groupe existe-t-il pour `sum(1)` ? `Q9(`k`)` produit le même résultat que `Q7(`k`)`, le groupe pertinent pour une constante est donc le plus interne : T_2.

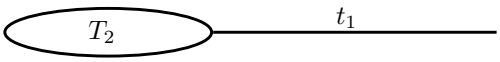
Constat 2. *Dans le même environnement,* `1+0*a2` *est égal à* `1` *et* `1+0*a1` *non puisque ces expressions sont calculées dans des contextes d'évaluations différents : sous un agrégat, en SQL, les égalités arithmétiques usuelles ne sont plus valides !*

Poursuivons notre analyse avec `Q10(`k`)` : qu'en est-il si les expressions `1+0*a1` et `1+0*a2` sont évaluées dans le même environnement où `1+0*a1` et `1+0*a2` apparaissent sous des agrégats *distincts* ? Il n'y a aucun groupe unique et naturellement pertinent. Quand $k \notin \{7, 12\}$ $[\![$`Q10(`k`)`$]\!]$ est vide alors que $[\![$`Q10(7)`$]\!] = \{|$`(a1=3);(a1=4)`$|\}$ et $[\![$`Q10(12)`$]\!] = \{|$`(a1=1);(a1=2)`$|\}$. Les deux expressions `1+0*a1` et `1+0*a2` ont été évaluées *indépendemment*, la première dans un contexte où T_1 a été scindé, la seconde dans un contexte ou c'est T_2 qui a été scindé.

Constat 3. *SQL autorise deux sous expressions d'une* même *expression à être évaluées dans différents contextes ! Ce qui est peu orthodoxe.*

Qu'advient il lorsque `1+0*a1` et `1+0*a2` apparaissent sous le *même* agrégat comme illustré par `Q11(`k`)` ? Pour k=2, $[\![$`Q11(2)`$]\!]$ vaut $\bigcup_{i=1}^{i=4} \{|$`(a1=i)`$|\}$, autrement $[\![$`Q11(`k`)`$]\!]$ est vide.

Constat 4. *T_2, le groupe pertinent le plus interne, a été scindé. T_1 a été aplati en un de ses éléments t_1 puisque seule sa partie homogène,* `a1`*, est utilisée par l'évaluation.*



Le lecteur aura noté que, jusqu'alors, les expressions présentes sous un agrégat mettent uniquement en jeu des attributs groupants. Que fait SQL si tel n'est pas le cas ? `Q12(`k`)` qui contient `sum(1+0*a1+0*b2)` se comporte exactement comme `Q11(`k`)`. `Q13(`k`)` contient `sum(1+0*b1+0*b2)`, conformément au Standard elle est *mal formée* et n'est pas évaluée. La raison réside dans le fait que deux attributs non groupants provenant de deux niveaux d'imbrication différents apparaissent sous l'agrégat. La dernière requête de ce bestiaire, `Q14(`k`)`, contenant l'expression `sum(1+0*b1+0*a2)`, est aussi mal formée et n'est pas évaluée. Cependant, nous aurions pu penser qu'elle eût pu être acceptée et évaluée dans le contexte suivant :

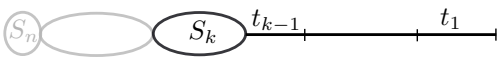
2.3 En résumé

Nous synthétisons au travers des définitions précisées d'environnements complétés et de contexte d'évaluation, les leçons tirées au cours de notre périple au coeur de SQL.

Environnements complétés Un environnement complété, $\mathscr{E} = [S_n; ...; S_1]$, est une pile de *tranches*. Chaque tranche correspond à un *niveau d'imbrication* i, le niveau le plus interne étant en tête de pile et de la forme $S = (A, G, T)$, où A (noté également $A(S)$) contient les attributs relevant de ce niveau d'imbrication, c'est à dire les noms introduits dans la sous requête à ce niveau [1] ; G les expressions de groupement introduites dans la clause `group by` (noté également $G(S)$) ; et T une liste non vide de n-uplets [2] (noté le cas échéant $T(S)$). Quand cela s'avérera utile nous adopterons la notation $\mathscr{E} = (A, G, T) :: \mathscr{E}'$ pour mettre en évidence la tête.

Contextes d'évaluation Lorsque e est une constante, la liste des n-uplets $T(S_n)$ provient de la tranche la plus interne de l'environnement $\mathscr{E} = [S_n; ...; S_1]$. Dans le cas où tous les attributs de e sont introduits au même niveau i, la liste est simplement $T(S_i)$. Lorsque les attributs de e appartiennent à au moins deux niveaux distincts, le plus interne (*i.e.*, celui d'indice plus élevé) étant S_k, nous sommes confrontés à deux cas :

- soit l'expression n'est pas bien formée (*cf* `Q13` et `Q14`), car e contient une expression de $T(S_j), j < k$ qui n'est pas groupée.
- soit l'expression e est exactement construite avec les attributs correspondant au k-ième niveau et les expressions groupantes [3] de niveaux externes $k-1, \ldots, 1$. Dans ce cas, soit t_j le n-uplet représentant pour chaque $T(S_j)$, $j < k$, la liste des n-uplets pertinents est faite de la concaténation $(t; t_{k-1}; \ldots; t_1)$, où t varie dans $T(S_k)$.



Nous présentons dans la section qui suit SQL_{Coq}, l'internalisation en Coq du fragment de SQL considéré, sa syntaxe ainsi que sa sémantique mécanisées.

3 SQL_{Coq} : une mécanisation de SQL en Coq

La syntaxe de SQL_{Coq} est présentée Figure 2, Figure 3 et Figure 4 où la partie gauche représente la syntaxe abstraite la droite son implantation en Coq.

1. Pour un `select from ...` ce sont les noms du `select`.
2. Quand la clause `group by` est présente c'est un groupe homogène autrement c'est un unique n-uplet.
3. Celle du `group by` pour ce niveau ; lorsqu'il n'y a pas de `group by` tous les attributs du niveau sont autorisés.

function ::= + | - | * | / | ... | *user defined fun*
aggregate ::= sum | avg | min | ... | *user defined ag*
value ::= string val | integer val | bool val | NULL
e^f ::= value | attribute | function($\overline{e^f}$)
e^a ::= e^f | aggregate(e^f) | function($\overline{e^a}$)

```
Inductive value : Set :=
  | String : string → value
  | Integer : Z → value
  | Bool : bool → value
  | NULL : value.
Inductive funterm : Type :=
   | F_Constant : value → funterm
   | F_Dot : attribute → funterm
   | F_Expr : symb → list funterm → funterm.
Inductive aggterm : Type :=
   | A_Expr : funterm → aggterm
   | A_agg : aggregate → funterm → aggterm
   | A_fun : symb → list aggterm → aggterm.
```

FIGURE 2: Expressions.

formula ::=
| formula (**and** | **or**) formula
| **not** formula
| **true**
| $p(\overline{e^a})$ $p \in$ *predicate*
| $p(\overline{e^a}, (\textbf{all} \mid \textbf{any})$ dom$)$ $p \in$ *predicate*
| $\overline{e^a}$ **as** attribute **in** dom
| **exists** dom

```
Inductive conjunct : Type := And | Or.
Inductive quantifier : Type := All | Any.
Inductive select : Type :=
    Select_As : aggterm → attribute → select.
Inductive formula (dom : Type): Type :=
  | Conj : conjunct → formula dom → formula dom → formula dom
  | Not : formula dom → formula dom
  | True : formula dom
  | Pred : predicate → list aggterm → formula dom
  | Quant : list aggterm → predicate → quantifier → dom →
            formula dom
  | In : list select → dom → formula dom
  | Exists : dom → formula dom.
```

FIGURE 3: Formules, paramétrées par un domaine fini d'interprétation dom.

3.1 SQL_{Coq} : syntaxe

Nous supposons donnés des attributs, des fonctions, des agrégats. Les valeurs peuvent être des chaînes, des entiers et des booléens ainsi que la valeur spéciale NULL. Les expressions sont usuellement construites à partir des valeurs, tout d'abord sans agrégats e^f, et ensuite avec e^a. Les formules en SQL sont similaires aux formules de la logique du premier ordre à cela près qu'elles sont interprétées dans un domaine fini qui est syntaxiquement référencé par dom sur la Figure 3. Ces mêmes formules seront également utilisées dans le contexte de l'algèbre SQL_{Alg}.

SQL_{Coq} reflète la syntaxe de SQL. Cependant le programmeur SQL averti aura noté quelques différences. Tout d'abord, à des fins d'uniformité, nous imposons d'avoir l'intégralité du bloc `select from where group by having` (les clauses `where` et `group by having` ne sont plus option-

select_item ::= $*$ | $\overline{e^a \textbf{ as } \text{attribute}}$
query ::=
| table
| query (**union** | **intersect** | **except**) query
| **select** select_item
from from_item
where formula
group by $\overline{e^f}$
having formula
from_item ::= query($\overline{\text{attribute as attribute}}$)

```
Inductive select_item : Type :=
  Select_Star | Select_List : list select → select_item.
Inductive att_renaming : Type :=
   | Att_As : attribute → attribute → att_renaming.
Inductive att_renaming_item : Type :=
   | Att_Ren_Star
   | Att_Ren_List : list att_renaming → att_renaming_item.
Inductive group_by : Type :=
   Group_Fine | Group_By : list funterm → group_by.
Inductive set_op : Type := Union | Intersect | Except.
Inductive query : Type :=
  | Table : relname → query
  | Set : set_op → query → query → query
  | Select : (* select *) select_item  →
             (* from *) list from_item →
             (* where *) formula query →
             (* group by *) group_by →
             (* having *) formula query → query
with from_item : Type :=
 | From_Item : query → att_renaming_item → sql_from_item.
```

FIGURE 4: Syntaxe de SQL_{Coq}

nelles). Lorsque la clause `where` est absente en SQL, elle est forcée à `true`. La table ci dessous résume la manière dont le parseur prend en compte les différents cas pour traduire l'absence ou la présence de `group by` en une construction de groupement en SQL_{Coq}.

SQL			SQL_{Coq}
`aggregate` (in `select`)	`group by`	`having`	
?	`g`	?	`Group_By g`
✓	✗	?	`Group_By nil`
?	✗	✓	`Group_By nil`
✗	✗	✗	`Group_Fine`

La construction `Group_Fine` correspond à la partition la plus fine[4] et diffère de `Group_By` $[a_1, \ldots, a_n]$ où $[a_1, \ldots, a_n]$ est la liste de labels de la requête courante. `Group_By` $[a_1, \ldots, a_n]$ est utilisé en SQL_{Coq} pour encoder le `distinct` de SQL. `Group_By nil` correspond à la partition grossière. Nous imposons également un renommage obligatoire et explicite des attributs quand $*$ n'est pas employé. Ainsi, dans notre syntaxe, `select a, b from t;` s'exprime par `select a as a, b as b from (table t[*]) where true group by Group_Fine having true`. Un dernier point, plus subtil, qu'il convient de mentionner est la distinction faite entre e^f et e^a. Toutes deux sont des expressions mais les premières sont construites uniquement avec des fonctions ($\mathfrak{fn}$) et sont évaluées sur des *n-uplets* tandis que les secondes utilisent des agrégats sans imbrication[5] ($\mathfrak{ag}$) et sont dans ce cas évaluées sur des *collections de n-uplets*. Nous utilisons le même langage pour les formules qu'elles apparaissent dans la clause `where` (portant sur un n-uplet) ou `having` (portant sur des collections de n-uplets) simplement en identifiant chaque n-uplet avec le singleton lui correspondant. Enfin, nous n'autorisons pas les alias pour les requêtes ce cas étant pris en compte par un renommage des attributs.

$$
\begin{array}{lll}
[\![c]\!]^{\mathrm{f}}_{\mathscr{E}} & = \mathrm{c} & \text{si } c \text{ est une value} \\
[\![a]\!]^{\mathrm{f}}_{[]} & = \textbf{default} & \text{si } a \text{ est un attribut} \\
[\![a]\!]^{\mathrm{f}}_{(A,G,[])::\mathscr{E}} & = [\![a]\!]^{\mathrm{f}}_{\mathscr{E}} & \\
[\![a]\!]^{\mathrm{f}}_{(A,G,t::T)::\mathscr{E}} & = t.\mathrm{a} & \text{si } a \in \ell(t) \\
[\![a]\!]^{\mathrm{f}}_{(A,G,t::T)::\mathscr{E}} & = [\![a]\!]^{\mathrm{f}}_{\mathscr{E}} & \text{si } a \notin \ell(t) \\
[\![\mathfrak{fn}(\overline{e})]\!]^{\mathrm{f}}_{\mathscr{E}} & = [\![\mathfrak{fn}]\!]_f(\overline{[\![e]\!]^{\mathrm{f}}_{\mathscr{E}}}) & \\
 & & \text{si } \mathfrak{fn} \text{ est une function,} \\
 & & \text{et } \overline{e} \text{ est une liste d'expressions simples}
\end{array}
$$

```
Definition env_type :=
   list (list attribute * group_by * list tuple).
Fixpoint interp_dot env (a : attribute) :=
 match env with
  | nil ⇒ default_value a
  | (sa, gb, nil) :: env' ⇒ interp_dot env' a
  | (sa, gb, t :: l) :: env' ⇒
     if a inS? labels t then (dot t a) else interp_dot env' a
   end.
Fixpoint interp_funterm  env t :=
 match t with
  | F_Constant c ⇒ c
  | F_Dot a ⇒ interp_dot env a
  | F_Expr f l ⇒
     interp_symb f (map (fun x ⇒ interp_funterm env x) l)
  end.
```

FIGURE 5: Sémantique des expressions simples.

3.2 SQL_{Coq} : sémantique

Cette section est le reflet en Coq des constats établis en Section 2. Étant donné un n-uplet t nous notons $\ell(t)$ les attributs présents dans t. Nous supposons l'existence d'une instance $[\![\_]\!]_{db}$ de la base de données définie comme une fonction des noms de relations vers des *multi-ensembles* de n-uplets[6] ainsi que d'interprétations prédéfinies, fixées $[\![\_]\!]_p$, pour les prédicats $\mathfrak{pr}$[7], *i.e.*, une fonction de vecteurs de valeurs vers les Booléens, $[\![\_]\!]_a$ et $[\![\_]\!]_f$ pour les agrégats

4. La partition consistant de collections de singletons, un singleton pour chaque n-uplet.
5. e^a est de la forme : `avg(a)` ; `sum(a+b)+3` ; `sum(a+b)+avg(c+3)` mais pas de la forme `avg(sum(c)+a)`
6. Ces multi-ensembles sont munis d'opérateurs similaires à ceux des listes, tels que empty, map, filter, etc.
7. $\mathfrak{pr}$ correspond à `<, in` etc.

$$\frac{c \in \mathscr{V}}{\mathbb{B}_u(G,c)} \qquad \frac{e \in G}{\mathbb{B}_u(G,e)} \qquad \frac{\bigwedge_{\overline{e}} \mathbb{B}_u(G,e)}{\mathbb{B}_u(G,\mathfrak{fn}(\overline{e}))}$$

$$\frac{\mathbb{B}_u((A \cup \bigcup_{(A',G,T)\in\mathscr{E}} G), e)}{\mathbb{S}_e(A,\mathscr{E},e)}$$

$$\frac{c \in \mathscr{V}}{\mathbb{F}_e(\mathscr{E},c) = \mathscr{E}} \qquad \frac{e \notin \mathscr{V}}{\mathbb{F}_e([],e) = \texttt{undefined}}$$

$$\frac{e \notin \mathscr{V} \quad \mathbb{F}_e(\mathscr{E},e) = \mathscr{E}'}{\mathbb{F}_e(((A,G,T)::\mathscr{E}),e) = \mathscr{E}'}$$

$$\frac{\mathbb{F}_e(\mathscr{E},e) = \texttt{undefined} \quad \mathbb{S}_e(A,\mathscr{E},e)}{\mathbb{F}_e(((A,G,T)::\mathscr{E}),e) = (A,G,T)::\mathscr{E}}$$

$$[\![\mathfrak{fn}(\overline{e})]\!]^{\mathrm{a}}_{\mathscr{E}} = [\![\mathfrak{fn}]\!]_f(\overline{[\![e]\!]^{\mathrm{a}}_{\mathscr{E}}})$$

$$[\![\mathfrak{ag}(e)]\!]^{\mathrm{a}}_{\mathscr{E}} = [\![\mathfrak{ag}]\!]_a \left(\overline{[\![e]\!]^{\mathrm{f}}_{((A,G,[t])::\mathscr{E}')}}\right)_{t\in T} \text{ ssi } \mathbb{F}_e(\mathscr{E},e) = (A,G,T)::\mathscr{E}'$$

```
Fixpoint (* (𝔹_u(G, f)) *) is_built_upon G f :=
 match f with
  | F_Constant _ ⇒ true
  | F_Dot _ ⇒ f inS? g
  | F_Expr s l ⇒ (f inS? G) || forallb (is_built_upon G) l
 end.
Definition (* (𝕊_e(la, env, f)) *) is_a_suitable_env la env f :=
  is_built_upon
   (map (fun a ⇒ F_Dot a) la ++
   flat_map (fun slc ⇒ match slc with (_, G, _) ⇒ G end) env)
   f.
Fixpoint (* (𝔽_e(env, f)) *) find_eval_env env f :=
 match env with
  | nil ⇒ if is_built_upon nil f then Some nil else None
  | (la1, g1, l1) :: env' ⇒
    match find_eval_env env' f with
     | Some _ as e ⇒ e
     | None ⇒
       if is_a_suitable_env la1 env' f then Some env else None
    end
 end.
Fixpoint interp_aggterm env (ag : aggterm) :=
  match ag with
  | A_Expr ft ⇒ (* simple expression without aggregate *)
     interp_funterm env ft
  | A_fun f lag ⇒
    (* simple recursive call in order to evaluate independently
     the sub-expressions when the top symbol is a function *)
    interp_symb f (List.map (fun x ⇒ interp_aggterm env x) lag)
  | A_agg ag ft ⇒
    let env' :=
        if is_empty (att_of_funterm ft)
        then (* expression under the aggregate is a constant *)
             Some env
        else (* find the outermost suitable level *)
             find_eval_env env ft in
    let lenv :=
        match env' with
          | None | Some nil ⇒ nil
          | Some ((la1, g1, l1) :: env'') ⇒
              (* outermost group is split into *)
               map (fun t1 ⇒ (la1, g1, t1 :: nil) :: env'') l1
        end in
    interp_aggregate ag
         (List.map (fun e ⇒ interp_funterm e ft) lenv)
  end.
```

FIGURE 6: Sémantique des expressions complexes (avec agrégats).

$\mathfrak{ag}$ et fonctions $\mathfrak{fn}$ respectivement [8]. Comme nous l'avons montré en Section 2, les expressions (complexes) apparaissant au sein de sous-requêtes potentiellement corrélées sont évaluées dans un environnement découpé en tranches, $\mathscr{E} = [S_n; ...; S_1]$ (ou $\mathscr{E} = (A,G,T) :: \mathscr{E}'$), le niveau le plus interne, n, correspondant à la première tranche. L'évaluation d'une entité syntaxique e de type x dans l'environnement $\mathscr{E}$ sera notée $[\![e]\!]^{\mathrm{x}}_{\mathscr{E}}$ (où x est f pour les expressions seulement construites avec des fonctions, a pour les expressions construites également avec des agrégats, b pour les formules et q pour les requêtes).

La sémantique des expressions simples est donnée Figure 5. La sémantique des expressions complexes, présentée Figure 6, mérite que l'on s'y attarde. Lorsque l'expression est gardée par un symbole de fonction, $\mathfrak{fn}(\overline{e})$, une simple descente récursive suffit. Lorsque l'expression est de la forme $\mathfrak{ag}(e)$, conformément à ce qui a été décrit en Section 2, il faut trouver le niveau d'imbrication adéquat afin de construire le groupe devant être scindé. Puis construire la liste de valeurs en évaluant e, et enfin évaluer $\mathfrak{ag}$ sur cette liste. Pour l'environnement $\mathscr{E}$=$[S_n; \ldots S_1]$, i est un niveau adéquat, exprimé par $\mathbb{S}_e(A(S_i), [S_{i-1}; \ldots; S_1], e)$ sur la Figure 6 dès lors que e est

8. a correspond à `sum`, `count` etc. et f : `+`,`*`,`-` etc.

$$
\begin{array}{lcl}
[\![f_1 \textbf{ and } f_2]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![f_1]\!]^{\mathrm{b}}_{\mathscr{E}} \wedge [\![f_2]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![f_1 \textbf{ or } f_2]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![f_1]\!]^{\mathrm{b}}_{\mathscr{E}} \vee [\![f_2]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![\textbf{not } f]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \neg [\![f]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![\textbf{true}]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
[\![\mathfrak{pr}(\overline{e_i})]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![\mathfrak{pr}]\!]_p(\overline{[\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}}}) \\
[\![\mathfrak{pr}(\overline{e_i}, \textbf{ all } q)]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{l}{\quad \text{ssi } [\![\mathfrak{pr}(\overline{e_i}, t)]\!]^{\mathrm{b}}_{\mathscr{E}} = \top^{\dagger} \text{ pour tout } t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\mathfrak{pr}(\overline{e_i}, \textbf{ any } q)]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{l}{\quad \text{ssi } [\![\mathfrak{pr}(\overline{e_i}, t)]\!]^{\mathrm{b}}_{\mathscr{E}} = \top^{\dagger} \text{ pour un } t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\overline{e_i \textbf{ as } a_i} \textbf{ in } q]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{l}{\qquad \text{si } (\overline{a_i = [\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}}}) \text{ appartient à}^{\dagger} \ [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\textbf{exists } q]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{l}{\qquad\qquad \text{ssi } [\![q]\!]^{\mathrm{q}}_{\mathscr{E}} \text{ est non vide}}
\end{array}
$$

[†] Voir le paragraphe sur les `NULL`' en Section 3.2.

```
Hypothesis I : env_type → dom → bagT.
Fixpoint eval_formula env (f : formula) : Bool.b B :=
 match f with
 | Sql_Conj a f1 f2 ⇒
     (interp_conj B a)
       (eval_formula env f1) (eval_formula env f2)
 | Sql_Not f ⇒ Bool.negb B (eval_formula env f)
 | Sql_True ⇒ Bool.true B
 | Sql_Pred p l ⇒ interp_pred p (map (interp_aggterm env) l)
 | Sql_Quant qtf p l sq ⇒
   let lt := map (interp_aggterm env) l in
   interp_quant B qtf
     (fun x ⇒ let la := Fset.elements _ (labels T x) in
             interp_pred p (lt ++ map (dot T x) la))
      (Febag.elements _ (I env sq))
 | Sql_In s sq ⇒
   let p := (projection env (Select_List s)) in
   interp_quant B Exists_F
    (fun x ⇒ match Oeset.compare (OTuple T) p x with
                | Eq ⇒ if contains_null p
                       then unknown else Bool.true B
                | _ ⇒ if (contains_null p || contains_null x)
                      then unknown else Bool.false B
                end)
       (Febag.elements _ (I env sq))
  | Sql_Exists sq ⇒
     if Febag.is_empty _ (I env sq)
     then Bool.false B else Bool.true B
 end.
```

FIGURE 7: Sémantique des formules.

construite avec $G = A(S_i) \cup \bigcup_{j<i} G(S_j)$ ce qui est exprimé par $\mathbb{B}_u(G, e)$ sur la Figure 6. Lorsque e est une constante, le niveau le plus interne est choisi (ici n), sinon le candidat pertinent le plus externe est choisi $\mathbb{F}_e(\mathscr{E}, e)$. La sémantique des formules donnée en Figure 7, s'appuie sur la sémantique des expressions. La syntaxe étant paramétrée par un domaine `dom`, de la même façon la sémantique est paramétrée par l'évaluation de ce domaine. Ce qui s'exprime dans notre développement Coq par : `Hypothesis I : env_type → dom → bagT.`, et par, $[\![\_]\!]^{\mathrm{q}}_{\_}$, dans la définition formelle.

La sémantique des requêtes, $[\![\_]\!]^{\mathrm{q}}_{\_}$ est détaillée sur la Figure 8. Les opérateurs ensemblistes sont munis, par défaut, d'une sémantique multi-ensembles (ce qui correspond à `union all`, `intersect all` etc en SQL). La sémantique strictement ensembliste pour `sq` = `q1 op q2`, est obtenue en appliquant l'opérateur d'élimination des doublons δ`(sq)` = `select * from sq` $\overline{(a_i \texttt{ as } a_i)}_{a_i \in \ell(sq)}$ `group by` ℓ`(sq)`. Le cas le plus complexe est celui du `select from where group by having`. Informellement, une première étape consiste à évaluer la partie `from` puis à filtrer le résultat au moyen du `where`. Plus précisément, vérifier qu'un n-uplet t satisfait la condition `where` w dans le contexte $\mathscr{E}$ s'opère ainsi : w est évaluée relativement à un unique environnement. Ceci implique que t et $\mathscr{E}$ doivent être combinés au sein d'un unique environnement, $\mathscr{E}'$, tel que $[\![w]\!]^{\mathrm{f}}_{\mathscr{E}'}$ corresponde à l'évaluation de w, où les attributs a éléments de $\ell(t)$ sont lié à $t.a$, et où les attributs a éléments de $\bigcup_{S \in \mathscr{E}} A(S)$ sont liés grâce à $\bigcup_{S \in \mathscr{E}} A(T)$. C'est exactement ce qui est fait lorsque $\mathscr{E}' = (\ell(t), [], [t]) :: \mathscr{E}$.

La collection (intermédiaire) de n-uplets obtenue est ensuite partitionnée au moyen des expressions de groupement du `group by` G, conduisant à l'obtention d'une collection de collections de n-uplets : les groupes. Pour rendre compte du caractère optionnel du `group by`, la partition la plus fine est utilisée ce qui est noté par `Group_Fine` dans le développement Coq.

La manière de filtrer les groupes relativement au `having` h est similaire à ce que nous avons décrit pour le `where`, mis à part que des expressions complexes peuvent être présentes dans h. Lors de l'évaluation d'une expression de la forme `ag`(e) pour un groupe T, tous les n-uplets du groupe sont indispensables ; lors de l'évaluation d'une expression simple, n'importe quel n-

$$
\begin{array}{lcl}
[\![tbl]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![tbl]\!]_{db} \\
\quad \text{si } tbl \text{ est une table} & & \\
[\![q_1 \ \textbf{union} \ q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \cup [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
[\![q_1 \ \textbf{intersect} \ q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \cap [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
[\![q_1 \ \textbf{except} \ q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \setminus [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
[\![\textbf{select} \ \overline{e_i \ \textbf{as} \ a_i} \ \textbf{from} \ \overline{f_i} \ \textbf{where} \ w & & \\
\qquad \textbf{group by} \ G \ \textbf{having} \ h]\!]^{\mathrm{q}}_{\mathscr{E}} = & & \\
\quad \left\{ \left| \overline{\left(a_i = [\![e_i]\!]^{\mathrm{a}}_{(\ell(T),G,T)::\mathscr{E}} \right)} \right| T \in \mathbb{F}_3 \right\} & & \\
\quad \text{si } \ F = \bowtie_i [\![f_i]\!]^{\mathrm{from}}_{\mathscr{E}} & & \\
\quad \text{et } F_1 = \left\{ t \in F \left| [\![w]\!]^{\mathrm{b}}_{(\ell(t),[],[t])::\mathscr{E}} = \top \right. \right\} & & \\
\quad \text{et } \mathbb{F}_2 \text{ est une partition}^{\dagger} \text{ de } F_1 \text{ selon } G & & \\
\quad \text{et } \mathbb{F}_3 = \left\{ T \in \mathbb{F}_2 \left| [\![h]\!]^{\mathrm{b}}_{(\ell(T),G,T)::\mathscr{E}} = \top \right. \right\} & & \\
[\![q(\overline{a_i \ \textbf{as} \ b_i})]\!]^{\mathrm{from}}_{\mathscr{E}} & = & \\
\qquad \{\!| \overline{(b_i = c_i)} \mid \overline{(a_i = c_i)} \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}} |\!\} & &
\end{array}
$$

[†]Voir le paragraphe sur les NULL's en Section 3.2.

```
Fixpoint eval_sql_query env sq {struct sq} :=
match sq with
 | Sql_Table tbl ⇒ instance tbl
 | Sql_Set o sq1 sq2 ⇒
   if sql_sort sq1 =S?= sql_sort sq2
   then Febag.interp_set_op _ o
          (eval_sql_query env sq1) (eval_sql_query env sq2)
   else Febag.empty _
 | Sql_Select s lsq f1 gby f2  ⇒
  let elsq :=
  (* evaluation of the from part *)
        List.map (eval_sql_from_item env) lsq in
  let cc :=
  (* selection of the from part by formula f1, with old names *)
     Febag.filter _
       (fun t ⇒
         Bool.is_true _
           (eval_sql_formula eval_sql_query (env_t env t) f1))
             (N_product_bag elsq) in
  (* computation of the groups grouped according to gby *)
  let lg1 := make_groups env cc gby in
  (* discarding groups according the having clause f2 *)
  let lg2 :=
    List.filter
     (fun g ⇒
        Bool.is_true _
          (eval_sql_formula eval_sql_query (env_g env gby g) f2))
      lg1 in
  (* applying outermost projection w.r.t. the select part s *)
   Febag.mk_bag BTupleT
     (List.map (fun g ⇒ projection (env_g env gby g) s) lg2)
 end
(* evaluation of the from part *)
with eval_sql_from_item env x :=
  match x with
  | From_Item sqj sj ⇒
    Febag.map BTupleT BTupleT
    (fun t ⇒ projection (env_t env t)
                        (att_renaming_item_to_from_item sj))
    (eval_sql_query env sqj)
    end.
```

FIGURE 8: Sémantique des requêtes.

uplet de T suffit car T est homogène relativement au critère de groupement G. Par conséquent, l'environnement correct pour filtrer le groupe T par h dans $\mathscr{E}$ est $(\ell(T), G, T) :: \mathscr{E}$. Enfin, le `select` est appliqué qui conduit à une collection de n-uplets : le résultat final.

À propos des NULL's Au niveau des expressions,, les NULL's se comportent comme des éléments absorbants relativement aux fonctions et sont ignorés pour les agrégats (à l'exception du `count(*)` où ils contribuent pour 1. Ceci est exprimé dans notre formalisation en imposant des contraintes sur $[\![\_]\!]_a$ et $[\![\_]\!]_f$. Pour les formules, une logique trivaluée est utilisée. L'évaluation de $\mathtt{pr}(\overline{e})$ dans l'environnement $\mathscr{E}$ vaut `unknown` ssi il existe e_i dans $\overline{e}$ telle que $[\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}} = \mathtt{NULL}$. La valeur `unknown` se distribue conformément aux règles classiques de la logique trivaluée. Les quantificateurs `all` et `any` sont vu respectivement comme la conjonction ou disjonction finie de la logique trivaluée. Enfin, $\overline{e \ \textbf{as} \ a}$ **in** q est évaluée comme la conjonction finie $\bigwedge \overline{e = t.a}$ où t varie dans $[\![q]\!]^{\mathrm{q}}$, ce qui induit que dès lors que e ou $t.a$ est évalué à NULL, $\bigwedge \overline{e = t.a}$ vaut `unknown`. Tôt ou tard, lors de l'évaluation des requêtes, l'évaluation des formules ayant conduit à `unknown` est convertie à false. Notons également que bien que NULL ne soit ni égal ni différent de NULL ou de quelqu'autre valeur dans le contexte de formules, NULL est bien égal à NULL pour le groupement. Ceci est pris en compte sur la Figure 8 par une définition minutieuse de partition et de `make_groups` dans le développement Coq.

4 SQL_{Alg} : une algèbre relationnelle mécanisée en Coq

4.1 L'algèbre relationnelle en bref

L'algèbre relationnelle (étendue) telle qu'elle est présentée dans les ouvrages de référence [8], est constituée des opérateurs σ (sélection), π (projection) and $\bowtie$ (jointure) étendue avec l'opérateur γ (groupement) complétée avec les opérateurs ensemblistes, intersection, union et différence. Nous nous concentrons sur les quatre premiers opérateurs dont nous rappelons la sémantique dans ce qui suit : $q := r \mid \sigma_f(q) \mid \pi_S(q) \mid q \bowtie q \mid \gamma_{g,ag}(q)$.

Les relations de base, r sont des expressions. L'opérateur de sélection permet de filtrer des collections de n-uplets, ne retenant que ceux qui satisfont la condition f. La sémantique de cet opérateur est $[\![\sigma_f(q)]\!] = \{t \mid t \in [\![q]\!] \wedge [\![f]\!]\{x \to t\}\}$ où $[\![f]\!]\{x \to t\}$ signifie "t satisfait la formule $[\![f]\!]$", x étant l'unique variable libre de $[\![f]\!]$.

L'opérateur de projection, de la forme π_S, opère sur toutes les expressions, q, dont la sorte contient le sous ensemble d'attributs S. Sa sémantique est donnée par $[\![\pi_S(q)]\!] = \{t|_S \mid t \in [\![q]\!]\}$ où la notation $t|_S$ représente la restriction du n-uplet t aux seuls attributs de S.

L'opérateur de jointure, noté $\bowtie$, prend en argument deux expressions q_1 et q_2 dont les sortes respectives sont V et W, et permet de combiner les n-uplets provenant de chaque opérande. Sa sémantique est donnée par $[\![q_1 \bowtie q_2]\!] = \{t \mid \exists v \in [\![q_1]\!], \exists w \in [\![q_2]\!], t|_V = v \wedge t|_W = w\}$.

Le dernier opérateur, $\gamma_{g,ag}$, est défini comme suit dans [8], « *operator $\gamma_{g,ag}$ partitions the tuples of q into groups. Each group consists of all tuples having one particular assignment of values to the grouping attributes in g. If there are no grouping attributes, the entire relation q is one group. For each group, one tuple consisting of the grouping attributes' values for that group and the aggregations, over all tuples of that group, for the aggregated attributes in ag is produced* ». Sa définition formelle est donnée en Figure 9 où f vaut true.

4.2 SQL_{Alg} syntaxe et sémantique

L'algèbre présentée dans [8], ne rend compte ni des formules de la clause `having` ni des expressions complexes (le groupement s'opère seulement avec des attributs et les agrégats ne sont appliqués qu'à un seul attribut) ni des environnements. Afin d'accueillir SQL, notre modélisation est plus expressive autorisant le groupement à partir d'expressions simples et permettant l'utilisation d'expressions complexes e^a dans les projections. Enfin, de sorte à considérer les conditions du `having`, qui opèrent directement sur des groupes, SQL_{Alg} étend l'algèbre décrite dans [8] en ajoutant un paramètre supplémentaire à γ : la condition de la clause `having`.

$$\begin{aligned} \text{Q} ::= {} & \mathit{table} \mid \text{Q } (\textbf{union} \mid \textbf{intersect} \mid \textbf{except}) \text{ Q} \mid \text{Q} \bowtie \text{Q} \\ & \mid \pi_{(\overline{e^a \textbf{ as attribute}})}(\text{Q}) \mid \sigma_{\text{formula}}(\text{Q}) \mid \gamma_{(\overline{e^a \textbf{ as attribute}}, \overline{ef}, \text{formula})}(\text{Q}) \end{aligned}$$

Nous donnons en Figure 9 la sémantique de SQL_{Alg}. Les expressions (simples et complexes) ainsi que les formules (dont paramètre de domaine `dom` est ici celui des requêtes algébriques) sont partagées avec SQL_{Coq}. Pour définir la sémantique des expressions, la notion d'environnement est nécessaire pour les mêmes raisons qu'en SQL_{Coq} : prendre en compte la corrélation. Ainsi, les environnements de SQL_{Alg} sont les mêmes qu'en SQL_{Coq}. Il faut noter que $\bowtie$ est le véritable opérateur de jointure naturelle et que γ peut être vu comme une version dégénérée de `select from where group by having`, dans laquelle la condition `where` est absente (ou vaut `true`). Nous sommes, à présent, en mesure de relier, formellement, SQL_{Coq} et SQL_{Alg}.

4.3 $\text{SQL}_{\text{Coq}} \equiv \text{SQL}_{\text{Alg}}$

La Figure 10, présente $\mathbb{T}^{\text{q}}(\_)$ une traduction de SQL_{Coq} vers SQL_{Alg}, ainsi que la traduction inverse $\mathbb{T}^{\text{Q}}(\_)$. Toutes deux s'appuient sur les traductions auxiliaires ($\mathbb{T}^{\text{f}}(\_)$, resp. $\mathbb{T}^{\text{F}}(\_)$) qui effectuent une simple traversée des formules en traduisant les requêtes qu'elles contiennent.

$$\begin{array}{l}
[\![tbl]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![tbl]\!]_{db} \qquad \text{si } tbl \text{ est une table} \\
[\![q_1 \ \mathbf{union} \ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \cup [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} \\
[\![q_1 \ \mathbf{intersect} \ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \cap [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} \\
[\![q_1 \ \mathbf{except} \ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \setminus [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}}
\end{array}$$

$$[\![q_1 \bowtie q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = \left\{ \left| \left(\overline{a_i = c_i}, \overline{b_j = d_j}\right) \right| \begin{array}{l} (\overline{a_i = c_i}) \in [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \wedge \\ (\overline{b_j = d_j}) \in [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} \wedge \\ (\forall\, i, j, \quad a_i = b_j \implies c_i = d_j) \end{array} \right. \right\}$$

$$\begin{array}{l}
[\![\pi_{(\overline{e_i \ \mathbf{as} \ a_i})}(q)]\!]^{\mathrm{Q}}_{\mathscr{E}} = \{\!|(\overline{a_i = [\![e_i]\!]^{\mathrm{a}}_{(\ell(t),[],[t])::\mathscr{E}}}) \mid t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}|\!\} \\
[\![\sigma_f(q)]\!]^{\mathrm{Q}}_{\mathscr{E}} = \{\!|t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}} \mid [\![f]\!]^{\mathrm{b}}_{(\ell(t),[],[t])::\mathscr{E}} = \top|\!\}
\end{array}$$

$$[\![\gamma_{(\overline{e_j \ \mathbf{as} \ a_j}, \overline{e_i}, f)}(q)]\!]^{\mathrm{q}}_{\mathscr{E}} = \left\{ \left| \overline{\left(a_j = [\![e_j]\!]^{\mathrm{a}}_{(\ell(T),\overline{e_i},T)::\mathscr{E}}\right)} \right| T \in \mathbb{F}_3 \right\}$$

si $\mathbb{F}_2$ est une partition de $[\![q]\!]^{\mathrm{Q}}_{\mathscr{E}}$ selon $\overline{e_i}$ et $\mathbb{F}_3 = \left\{ \left| T \in \mathbb{F}_2 \left| [\![f]\!]^{\mathrm{b}}_{(\ell(T),\overline{e_i},T)::\mathscr{E}} = \top \right. \right| \right\}$

FIGURE 9: Sémantique de $\mathrm{SQL}_{\mathrm{Alg}}$

$$\begin{array}{l}
\mathbb{T}^{\mathrm{q}}(tbl) = tbl \\
\mathbb{T}^{\mathrm{q}}(q_1 \ \mathbf{op} \ q_2) = \mathbb{T}^{\mathrm{q}}(q_1) \ \mathbf{op} \ \mathbb{T}^{\mathrm{q}}(q_2) \qquad \text{où } \mathbf{op} \in \{ \mathbf{union}, \mathbf{intersect}, \mathbf{except} \} \\
\mathbb{T}^{\mathrm{q}}(\mathbf{select} \ \overline{e_i \ \mathbf{as} \ a_i} \ \mathbf{from} \ \overline{f_i} \ \mathbf{where} \ w) = \pi_{(\overline{e_i \ \mathbf{as} \ a_i})}(\sigma_{\mathbb{T}^{\mathrm{f}}(w)}(\bowtie_i \overline{\mathbb{T}^{\mathrm{from}}(f_i)})) \\
\mathbb{T}^{\mathrm{q}}(\mathbf{select} \ \overline{e_i \ \mathbf{as} \ a_i} \ \mathbf{from} \ \overline{f_i} \ \mathbf{where} \ w \ \mathbf{group\ by} \ G \ \mathbf{having} \ h) = \gamma_{(\overline{e_i \ \mathbf{as} \ a_i}, G, \mathbb{T}^{\mathrm{f}}(h))}(\sigma_{\mathbb{T}^{\mathrm{f}}(w)}(\bowtie_i \overline{\mathbb{T}^{\mathrm{from}}(f_i)}))
\end{array}$$

$$\mathbb{T}^{\mathrm{from}}(\overline{q(a_i \ \mathbf{as} \ b_i)}) = \pi_{\overline{(a_i \ \mathbf{as} \ b_i)}}(\mathbb{T}^{\mathrm{q}}(q))$$

$$\begin{array}{l}
\mathbb{T}^{\mathrm{Q}}(tbl) = tbl \\
\mathbb{T}^{\mathrm{Q}}(q_1 \ \mathbf{op} \ q_2) = \mathbb{T}^{\mathrm{Q}}(q_1) \ \mathbf{op} \ \mathbb{T}^{\mathrm{Q}}(q_2) \qquad \text{où } \mathbf{op} \in \{ \mathbf{union}, \mathbf{intersect}, \mathbf{except} \} \\
\mathbb{T}^{\mathrm{Q}}(q_1 \bowtie q_2) = \mathbf{select} \ \overline{(a'_1 \ \mathbf{as} \ a_1)}_{a_1 \in \ell(q_1)}, \overline{a'_2 \ \mathbf{as} \ a_2}_{a_2 \in \ell(q_2) \setminus \ell(q_1)} \mathbf{from} \ [\mathbb{T}^{\mathrm{Q}}(q_1)(\overline{a_1 \ \mathbf{as} \ a'_1}); \mathbb{T}^{\mathrm{Q}}(q_2)(\overline{a_2 \ \mathbf{as} \ a'_2})] \\
\qquad \mathbf{where} \ \overline{(a'_1 = a'_2)}_{a_1 \in \ell(q_1), a_2 \in \ell(q_2), a_1 = a_2} \qquad \text{où } \overline{a'_1} \text{ et } \overline{a'_2} \text{ sont des noms frais} \\
\mathbb{T}^{\mathrm{Q}}(\pi_{(\overline{e \ \mathbf{as} \ a})}(q)) = \mathbf{select} \ (\overline{e \ \mathbf{as} \ a}) \ \mathbf{from} \ [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \ \mathbf{as} \ a})] \\
\mathbb{T}^{\mathrm{Q}}(\sigma_f(q)) = \mathbf{select} \ * \ \mathbf{from} \ [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \ \mathbf{as} \ a})] \mathbf{where} \ \mathbb{T}^{\mathrm{F}}(f) \\
\mathbb{T}^{\mathrm{Q}}(\gamma_{(\overline{e \ \mathbf{as} \ a}, G, f)}(q)) = \mathbf{select} \ (\overline{e \ \mathbf{as} \ a}) \ \mathbf{from} \ [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \ \mathbf{as} \ a})] \ \mathbf{group\ by} \ G \ \mathbf{having} \ \mathbb{T}^{\mathrm{F}}(f)
\end{array}$$

FIGURE 10: Traductions entre $\mathrm{SQL}_{\mathrm{Coq}}$ et $\mathrm{SQL}_{\mathrm{Alg}}$.

Les expressions, partagées par $\mathrm{SQL}_{\mathrm{Alg}}$ et $\mathrm{SQL}_{\mathrm{Coq}}$, sont inchangées par les traductions. Ces traductions sont correctes dès lors qu'elles sont appliquées sur des instances de bases de données et des requêtes « raisonnables » :

Définition 1. *Une instance de bases de données* $[\![\_]\!]_{db}$ *est* bien sortée *si et seulement si tous les n-uplets d'une même relation possèdent les mêmes labels :*
$\forall r, t_1, t_2, \quad t_1 \in [\![r]\!]_{db} \wedge t_2 \in [\![r]\!]_{db} \implies \ell(t_1) = \ell(t_2)$.

Définition 2. *Une requête* SQL_{Coq} *sq est* bien formée *si et seulement si tous les labels introduits dans la clause* `from` *sont disjoints deux à deux et si toutes ses sous-requêtes sont récursivement bien formées.*

$$\frac{}{\overline{\mathbb{W}^q(tbl)}} \ tbl \text{ est une table} \qquad \frac{\mathbb{W}^q(q_1) \quad \mathbb{W}^q(q_2)}{\mathbb{W}^q(q_1 \ \mathbf{union} \ q_2)} \qquad \frac{\mathbb{W}^q(q_1) \quad \mathbb{W}^q(q_2)}{\mathbb{W}^q(q_1 \ \mathbf{intersect} \ q_2)} \qquad \frac{\mathbb{W}^q(q_1) \quad \mathbb{W}^q(q_2)}{\mathbb{W}^q(q_1 \ \mathbf{except} \ q_2)}$$

$$\frac{\mathbf{disjoint}\overline{\{b_i\}_i} \quad \bigwedge_i \mathbb{W}^q(q_i) \quad \mathbb{W}^f(w) \quad \mathbb{W}^f(h)}{\mathbb{W}^q(\mathbf{select} \ s \ \mathbf{from} \ \overline{q_i(\overline{a_i \ \mathbf{as} \ b_i})} \ \mathbf{where} \ w \ \mathbf{group\ by} \ G \ \mathbf{having} \ h)}$$

$$\frac{\mathbb{W}^f(f_1)\quad \mathbb{W}^f(f_2)}{\mathbb{W}^f(f_1 \textbf{ and } f_2)} \qquad \frac{\mathbb{W}^f(f_1)\quad \mathbb{W}^f(f_2)}{\mathbb{W}^f(f_1 \textbf{ or } f_2)} \qquad \frac{\mathbb{W}^f(f)}{\mathbb{W}^f(\textbf{not } f)} \qquad \frac{}{\mathbb{W}^f(\textbf{true})} \qquad \frac{}{\mathbb{W}^f(\mathfrak{pr}(\overline{e}_i))}$$

$$\frac{\mathbb{W}^q(q)}{\mathbb{W}^f(\textbf{exists } q)} \qquad \frac{\mathbb{W}^q(q)}{\mathbb{W}^f(\mathfrak{pr}(\overline{e}_i, \textbf{all } q))} \qquad \frac{\mathbb{W}^q(q)}{\mathbb{W}^f(\mathfrak{pr}(\overline{e}_i, \textbf{any } q))} \qquad \frac{\mathbb{W}^q(q)}{\overline{e_i \textbf{ as } a_i} \textbf{ in } q}$$

Quand ces conditions sont remplies, SQL_{Coq} et SQL_{Alg} sont sémantiquement équivalents :

Théorème 1. *Soit* $[\![\_]\!]_{db}$ *une instance* bien sortée*, sq une requête* SQL_{Coq} *et aq une requête* SQL_{Alg} *:*

$$\begin{array}{c} \forall \mathscr{E}, sq, \mathbb{W}^q(sq) \implies [\![\mathbb{T}^q(sq)]\!]^Q_{\mathscr{E}} = [\![sq]\!]^q_{\mathscr{E}} \\ \forall \mathscr{E}, aq, \quad [\![\mathbb{T}^Q(aq)]\!]^q_{\mathscr{E}} = [\![aq]\!]^Q_{\mathscr{E}} \end{array}$$

La preuve procède par induction structurelle mutuelle sur les requêtes et formules en utilisant leurs tailles respectives comme mesure. Dans la preuve de correction de $\mathbb{T}^q(\_)$, l'hypothèse $\mathbb{W}^q(sq)$ assure que le produit Cartésien et la jointure naturelle coïncident. Cette hypothèse de bonne formation est essentielle pour assurer le résultat ce qui donne un éclairage intéressant au fait qu'en réalité la clause `from` de SQL se comporte vraiment comme un produit. L'hypothèse de bonne sorte assure une forme de typage faible : les n-uplets d'une même relation ont les mêmes attributs que ceux déclarés lors de la définition de la relation. Ceci est toujours le cas dans la réalité. Cette hypothèse permet de raisonner sur les attributs de manière globale, en calculant statiquement les labels de la requête pour évaluer celle-ci.

5 Conclusions

De longue date, la communauté bases de données s'est employée à définir une sémantique formelle pour SQL. Cet article propose une sémantique *exécutable, mécanisée* en Coq pour un fragment réaliste du langage. *La mécanisation* au sein d'un assistant à la preuve ouvre d'intéressantes perspectives. Les fonctions récursives, en Coq, devant être totales, une fois fixée la syntaxe, la sémantique doit être « totalement » définie : aucun détail ne peut être occulté. Tous les cas devant être considérés, ceci nous a permis de proposer le florilège de requêtes étranges de la Figure 1 qui pourraient servir de base à un « benchmark » sémantique pour tester différentes implantations du langage. Produire une sémantique *exécutable* permet de se convaincre de sa correction puisqu'il est alors possible de la confronter aux systèmes existants. Ce que nous avons fait en testant systématiquement le comportement de SQL_{Coq} vis à vis de celui de PostgreSQL et Oracle™.

Définir formellement la sémantique de SQL et la relier rigoureusement à l'algèbre relationnelle, nous a permis d'établir le premier résultat, à notre connaissance, d'équivalence entre les deux langages, recouvrant ainsi les équivalences algébriques qui fondent les optimisations opérées par le compilateur SQL. Ces équivalences sont formellement prouvées dans [3]. Quoique le sachant, nous avons eu la confirmation que, SQL ayant été initialement conçu comme un langage dédié n'ayant pas vocation à être Turing-complet, l'ajout, au cours du processus de standardisation, de traits nouveaux nécessaires, l'a inexorablement écarté de ses fondement élégants. En reliant formellement SQL et l'algèbre relationnelle nous souhaitions, humblement, rendre hommage aux pères fondateurs qui jetèrent les bases des SGBDR.

Dans une version préliminaire de ce travail nous avions proposé une sémantique purement ensembliste. Puis nous nous sommes attelées à la version multi-ensemble et avons été agréablement surprises de découvrir que cette évolution se passe sans heurts. En parfaite contradiction avec la croyance, répandue dans le monde des bases de données, que *la* difficulté consiste à doter SQL d'une sémantique multi-ensemble. La prise en compte des `NULL`'s est souvent considérée comme difficile. Ceci est dû au fait que SQL ne les traite pas uniformément selon le contexte. La

logique trivaluée nous a suffi à en rendre compte. Le véritable défi a consisté analyser scrupuleusement la gestion des environnements opérée par SQL en présence de requêtes imbriquées et corrélées et de combiner les quatre aspects afin de fournir une sémantique fidèle. Le document de standardisation ISO ne nous a guère aidées, Coq, en revanche, a été un maître intraitable dont l'aide nous fût des plus précieuses.

Notre objectif à long terme est l'obtention d'un compilateur de SQL vérifié. Le travail présenté dans cet article fourni un analyseur sémantique certifié, étape cruciale de la chaîne de compilation. Nous envisageons d'étendre le fragment considéré à la construction `order by`. Les aspects relatifs aux couches basses d'un moteur d'exécution de SQL sont présentés dans [4] qui propose une spécification et une implantation certifiée des opérateurs physiques. Il nous reste à traiter des aspects liés à l'optimiseur du compilateur.

Références

[1] T. Arvin. *Comparison of different SQL's implementations*, 2017.

[2] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In *SIGMOD Conference, Chicago, USA*, 2017.

[3] V. Benzaken, E. Contejean, and S. Dumbrava. A Coq Formalization of the Relational Data Model. In *23rd European Symposium on Programming (ESOP)*, 2014.

[4] V. Benzaken, É. Contejean, C. Keller, and E. Martins. A Coq formalisation of SQL's execution engines. In *Int. Conf. on Interactive Theorem Proving (ITP 2018)*, Oxford, UK, July 2018.

[5] S. Ceri and G. Gotlob. Translating SQL into relational algebra : Optimisation, semantics, and equivalence of SQL queries. *IEEE Trans., on Software Engineering*, SE-11 :324–345, April 1985.

[6] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL : Proving query rewrites with univalent SQL semantics. In *PLDI*. ACM, 2017.

[7] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Database Programming Languages, New York City, USA*, 1993.

[8] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[9] C. Gonzalia. Towards a formalisation of relational database theory in constructive type theory. In *RelMiCS*, 2003.

[10] C. Gonzalia. *Relations in Dependent Type Theory*. PhD thesis, Chalmers Göteborg Univ., 2006.

[11] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc., of the 26th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Beijing, China*, 2007.

[12] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1) :27–39, 2017.

[13] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[14] ISO/IEC. Information technology - database languages - SQL - part 2 : Foundation (SQL/foundation), 2006. Final Commitee Draft.

[15] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4) :363–446, 2009.

[16] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *ACM Int. Conf. POPL*, 2010.

[17] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3) :513–534, 1991.

[18] The Agda Development Team. *The Agda Proof Assistant Reference Manual*, 2010.

[19] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2010.

[20] The Isabelle Development Team. *The Isabelle Interactive Theorem Prover*, 2010.