Table des matières

Première partie : Contexte **1**

I Langages centrés données **3**
I.1 Evolution des bases de données 4
I.1.1 Paradigme navigationnel 5
I.1.2 Paradigme relationnel 7
I.1.3 Paradigmes imbriqués et paradigme objet 8
I.1.4 Paradigmes semi-structurés 8
I.1.5 NoSQL 9
I.2 L'algèbre relationnelle (RA) 9
I.2.1 Le modèle relationnel 9
I.2.2 Les opérateurs relationnels 12
I.3 Le langage SQL 14
I.3.1 Modèle de données de SQL 14
I.3.2 Le langage d'interrogation des données 15
I.3.3 Spécificités de SQL 18
I.3.3.1 Les valeurs nulles 18
I.3.3.2 Les requêtes corrélées 18
I.4 L'algèbre imbriquée (NRA) 21
I.4.1 Modèle de données 21
I.4.2 Les opérateurs de NRA 22
I.5 Comparaison entre RA, SQL et NRA 25
I.6 Conclusion 25

II Compilation certifiée en Coq **27**
II.1 Méthodes formelles 28
II.2 Assistants de preuve 28
II.2.1 Coq 29
II.3 Compilation certifiée correcte 30
II.3.1 Définitions 30
II.3.1.1 Compilation 30

II.3.1.2 Encodage et décodage 30
II.3.1.3 Mécanisation d'un langage 30
II.3.1.4 Compilateur algébrique 31
II.3.2 Correction d'un compilateur algébrique 31
II.3.2.1 Encodage et Décodage isomorphiques 31
II.3.2.2 Encodage homomorphique 32
II.3.2.3 Décodage homomorphique 32
II.3.3 Compilation certifiée en Coq d'un langage jouet 32
II.3.3.1 Le langage `ToyExp` 33
II.3.3.2 Instanciations de `ToyExp` 33
II.3.3.3 Définition et certification de `ToyExpCert` 33
II.3.3.4 Instanciation du compilateur `ToyExpCert` 34
II.4 Conclusion 35

III Formalisation de SQL et de NRA **39**
III.1 SqlCert 40
III.1.1 Présentation de SqlCert 40
III.1.2 SQL_{Alg} : Une extension de RA pour SQL 41
III.1.2.1 Représentation des données 42
III.1.2.2 Syntaxe 43
III.1.2.3 Exemples de requêtes 46
III.1.2.4 Les environnements de SQL_{Alg} 46
III.1.2.5 Sémantique 49
Sémantique des expressions sans agrégats 50
Sémantique des expressions avec agrégats 51
Sémantique des formules 52
Sémantique des requêtes 53
III.1.2.6 Preuve de concept : Une instance de la spécification 54
III.2 Q*Cert 57
III.2.1 Présentation de Q*Cert 57
III.2.2 NRA^e 57
III.2.2.1 Représentation des données 57
III.2.2.2 Syntaxe de NRA^e 58
III.2.2.3 Sémantique de NRA^e 60
III.3 Conclusion 62

Deuxième partie : Contribution **65**

IV De SQL_{Alg} vers NRA^e **67**
IV.1 Objectifs et présentation de la méthodologie 68
IV.1.1 Objectifs de la traduction 68
IV.1.2 Présentation générale de la traduction 69
IV.2 Encodage du modèle de données 70

IV.2.1 Encodage des valeurs . . . 71
IV.2.2 Encodage des noms d'attributs . . . 71
IV.2.3 Encodage de la logique trivaluée . . . 71
IV.2.4 Encodage des tuples . . . 72
IV.2.5 Encodage des collections . . . 73
IV.2.6 Récapitulatif de l'encodage des données . . . 74
IV.3 Encodage des instances de bases de données . . . 74
IV.4 Encodage et gestion des environnements . . . 75
IV.5 La traduction de SQL_{Alg} . . . 79
IV.5.1 Les expressions . . . 79
IV.5.1.1 Expressions simples . . . 80
IV.5.1.2 Expressions complexes . . . 83
IV.5.1.3 Listes d'expressions nommées . . . 85
IV.5.2 Les formules . . . 85
IV.5.2.1 Symboles de prédicat . . . 85
IV.5.2.2 Opérateurs NRA^e de la logique trivaluée . . . 87
IV.5.3 Les requêtes . . . 89
IV.5.4 Exemple de traduction . . . 92
IV.6 Réalisation des spécifications . . . 92
IV.6.1 Les valeurs nulles . . . 93
IV.6.2 Valeurs de la logique trivaluée . . . 94
IV.6.3 Les symboles de fonction . . . 95
IV.6.4 Les symboles d'agrégat . . . 96
IV.6.5 Les symboles de prédicat et les opérateurs logiques . . . 96
IV.7 Théorèmes de correction de la traduction . . . 98
IV.8 Conclusion . . . 101

Troisième partie : Discussion **103**

V Discussion **105**
V.1 Cas d'application : DBCert . . . 105
V.1.1 Présentation de DBCert . . . 105
V.1.2 Préservation de la sémantique . . . 106
V.1.3 Exemple . . . 106
V.1.4 Évaluation de DBCert . . . 107
V.1.4.1 AlaSQL . . . 108
V.1.4.2 *Benchmarks* . . . 108
V.1.4.3 Résultats . . . 109
V.1.4.4 Performances . . . 110
V.2 Travaux connexes . . . 110

Conclusion et perspectives 115

Références bibliographiques 116

Annexes 125

A Matériel supplémentaire 125

B Syntaxe d'une requête SELECT 127

C Sémantique opérationnelle de NRA^e 129

D Spécification paramétrant SQL$_{Alg}$ 131

Table des figures

I.1 Exemple d'une base de données hiérarchique. 7
I.2 Exemple d'une base de données relationnelle 11
I.3 Projection ensembliste vs. Projection multiensembliste 15
I.4 Syntaxe du fragment `select from where group by having` de SQL . . 16
I.5 Example de requêtes corrélées 19
I.6 Exemple d'une relation dans le modèle NRA 22
I.7 Exemple de l'application d'un `group by` (NRA) 24

II.1 Correction de compilation selon [Mosses, 1980] 32
II.2 Correction de compilation selon [Thatcher *et al.*, 1980] 32
II.3 Correction de compilation selon [Morris, 1973] 33
II.4 Spécification/Définition du langage `ToyExp` 34
II.5 Instanciations du langage `ToyExp` 35
II.6 Définition et certification de `ToyExpCert` 36
II.7 Instanciation du compilateur `ToyExpCert` 37

III.1 Syntaxe de SQL_{Coq} 40
III.2 Modèle générique paramétrant la formalisation Coq de SQL_{Alg} 43
III.3 Syntaxe des expressions SQL_{Alg} 44
III.4 Fonctions et agrégats génériques paramétrant les expressions SQL_{Alg} . . . 44
III.5 Syntaxe des formules de SQL_{Alg} 45
III.6 Prédicats génériques paramétrant les formules SQL_{Alg} 45
III.7 Syntaxe des requêtes SQL_{Alg} 45
III.8 Exemple d'un environnement de SQL_{Alg} 47
III.9 Sémantique de la fonction qui trouve le bon environnement d'évaluation d'une expression avec agrégat 52
III.10 Modèle de données de NRA^e 58
III.11 Définition du modèle de données de NRA^e sous Coq 58
III.12 Syntaxe de NRA^e 59

IV.1 Architecture de la traduction 70
IV.2 Spécification en Coq de l'encodage du modèle de donnée 72

IV.3 Spécification en Coq de l'encodage de la logique trivaluée 72
IV.4 Définition en Coq de l'encodage des tuples 73
IV.5 Définition en Coq de l'encodage des multiensembles de tuples 74
IV.6 Spécification et définition en Coq de l'encodage de l'instance de la base de données 75
IV.7 Extraction de la partie abstraite d'un environnement 76
IV.8 Définition en Coq de l'encodage de la partie concrète d'un environnement SQL_{Alg} 77
IV.9 Définition de la fonction `push` qui ajoute une tranche à l'environnement courant 78
IV.10 Spécification en Coq de la traduction des symboles de fonction 80
IV.11 Spécification en Coq de la traduction des expressions simples 82
IV.12 Spécification en Coq de la traduction des symboles d'agrégat 83
IV.13 Sémantique de la fonction qui trouve le bon environnement abstrait de traduction d'une expression avec agrégat 84
IV.14 Définition en Coq de la traduction des expressions nommées 85
IV.15 Spécification en Coq de la traduction des symboles de prédicat 86
IV.16 Spécification en Coq des opérateurs NRA^{e} de la logique trivaluée et de la traduction des requêtes SQL_{Alg} 88
IV.17 Traduction des formules SQL_{Alg} 89
IV.18 Traduction des requêtes SQL_{Alg} 91
IV.19 Spécification en Coq de tous les éléments nécessaires à la traduction des requête de SQL_{Alg} 92
IV.20 Représentation schématique des instances 93
IV.21 Réalisation de l'encodage des valeurs 94
IV.22 Réalisation de l'encodage de la logique trivaluée 95
IV.23 Définition de l'opérateur $\texttt{in}_B$ 99
IV.24 Diagramme de simulation de la traduction 99

V.1 Architecture du projet DBCert 106

Première partie : Contexte

Cette thèse se situe au carrefour de deux disciplines, les langages centrés données d'une part et les méthodes formelles de l'autre. Bien que les méthodes formelles aient apporté des garanties sur la correction des logiciels critiques dans différents champs de recherche et d'application, elles n'ont curieusement été que peu utilisées pour formaliser et étudier les langages et systèmes centrés données.

Parmi les travaux de recherche les plus aboutis qui associent ces deux domaines, SqlCert [Benzaken et Contejean, 2019] et Q*Cert [Auerbach *et al.*, 2017d] s'appuient sur l'assistant de preuve Coq pour formaliser et étudier les langages centrés données.

Notre travail de recherche jette le pont entre ces deux projets en proposant une traduction certifiée en Coq d'un langage proposé dans SqlCert vers un langage introduit dans Q*Cert.

Dans cette partie, nous présentons les langages centrés données en portant une attention particulière à ceux qui sont en lien avec cette thèse. Ensuite, nous introduisons la méthodologie utilisée à travers la présentation des méthodes formelles et de l'assistant de preuve Coq. Enfin, nous exposons les deux projets cités précédemment qui sont à la base de notre travail.

Cette partie contextualise notre thèse. Par souci pédagogique, l'existant est cité au fur et à mesure de l'introduction et l'explication des différents concepts et notions. Ce choix est motivé par le positionnement de notre thèse à la confluence de plusieurs champs de recherche. Des travaux connexes complémentaires sont discutés dans V.2. Travaux connexes.

Chapitre

I

Langages centrés données

Cette thèse s'inscrit dans le cadre de la formalisation de langages centrés données. Ces langages sont dédiés aux traitements de données. Ils sont le plus souvent déclaratifs [1]. Dans ce chapitre, nous nous intéressons à l'évolution des modèles de données et des langages qui les manipulent. Nous portons particulièrement notre attention sur l'algèbre relationnelle [Codd, 1970] qui est le langage référence, puis sur SQL qui est sémantiquement équivalent [2] au langage source de la traduction certifiée que nous proposons dans cette thèse. Enfin, nous présentons également le langage NRA, qui est, quant à lui, sémantiquement équivalent [3] au langage cible de notre traduction.

I.1 Evolution des bases de données **4**
I.1.1 Paradigme navigationnel 5
I.1.2 Paradigme relationnel 7
I.1.3 Paradigmes imbriqués et paradigme objet 8
I.1.4 Paradigmes semi-structurés 8
I.1.5 NoSQL 9
I.2 L'algèbre relationnelle (RA) **9**
I.2.1 Le modèle relationnel 9
I.2.2 Les opérateurs relationnels 12
I.3 Le langage SQL **14**
I.3.1 Modèle de données de SQL 14
I.3.2 Le langage d'interrogation des données 15
I.3.3 Spécificités de SQL 18
I.3.3.1 Les valeurs nulles 18
I.3.3.2 Les requêtes corrélées 18
I.4 L'algèbre imbriquée (NRA) **21**
I.4.1 Modèle de données 21
I.4.2 Les opérateurs de NRA 22
I.5 Comparaison entre RA, SQL et NRA **25**
I.6 Conclusion **25**

1. Un langage déclaratif *décrit* le résultat attendu sans détailler comment l'obtenir.
2. Equivalence prouvée en Coq.
3. Equivalence prouvée en Coq.

I.1. Evolution des bases de données

On dit que l'Histoire commence avec l'invention de l'écriture qui a permis à l'homme de prendre conscience de la possibilité de *produire* et de *conserver* une substance *immatérielle*, d'abord des inventaires ou des registres, puis des histoires, pour arriver à ce que représente l'écriture aujourd'hui. De la même manière, on peut dire que l'invention de la *tabulatrice* en 1889 pourrait être vue dans quelque temps comme la fin du *préinformatique* et la naissance de l'ère du *traitement automatique de l'information*.

La tabulatrice a été inventée par Hermann Hollerith en 1889. Il s'agit d'une machine électromécanique qui utilise des cartes perforées capables de traiter des données statistiques [4]. Hollerith utilisa les cartes perforées [5] pour *stocker* des tableaux statistiques et inventa ainsi le premier support de données. Il fabriqua également une machine avec le principe suivant : une information est représentée par l'absence ou la présence d'un trou dans un support. Le trou est détecté mécaniquement et l'information est comptabilisée électriquement. La tabulatrice a eu un franc succès avec les résultats obtenus dans le cadre du recensement national de la population des États-Unis. En 1896, Hollerith fonda l'entreprise *TMC* [6] qui deviendra *IBM* en 1924.

Le modèle le plus courant de cartes perforées contient 12 *lignes* et 80 *colonnes*. Un croisement entre une ligne et une colonne peut être perforé ou pas. La présence de trou correspond à une *valeur* pour une *information*. Les lignes de 0 à 9 représentent les chiffres de 0 à 9 et les lignes 11 et 12 [7] permettent de représenter les lettres. Par exemple, la valeur d'une colonne qui a une perforation uniquement sur la ligne 1 est 1 et la valeur d'une colonne qui a deux perforations, l'une sur la ligne 1 et l'autre sur la ligne 12 est *A*. Les cartes perforées étaient souvent *étiquetées* par une description générale du contenu et par des *étiquettes* au niveau des *champs* d'informations ou des valeurs. Dans une carte perforée, ce n'est pas les données qui sont stockées, mais plutôt une *représentation* de ces données. A cette époque, le stockage et la représentation des données étaient indissociables.

Jusqu'au milieu des années cinquante, les ordinateurs servaient principalement à effectuer des calculs complexes et parfois, plus rarement, à traiter des informations encodées dans des fichiers [8].

C'est l'apparition, en 1956, du premier disque dur, l'*IBM 350*, qui a fait prendre une nouvelle dimension à l'ordinateur. Il était dès lors possible, dans des conditions raisonnables, de collecter et de stocker des informations, de les consulter, de les modifier et de les traiter. C'est le premier germe semé dans ce que deviendra le champ des bases de données.

Jusqu'au début des années soixante, les informations étaient encodées et stockées sous

4. Pour le recensement de 1890, qui représentait une quantité de travail colossale pour les statisticiens.
5. Inventées par Jacquard en 1801 pour les métiers à tisser.
6. Tabulating Machine Company.
7. Ces deux lignes ce trouvent en haut de la carte.
8. Ces fichiers étaient stockés dans des cartes perforées.

forme d'*enregistrements* [9] dans des fichiers. Ceci a fait rapidement apparaître certaines limitations qu'il fallait vite lever, parmi celles-ci :

— l'accès séquentiel aux enregistrements contenus dans un fichier ralentissait énormément la consultation d'informations ciblées ;

— la modification et la gestion des enregistrements demandaient parfois des efforts fastidieux d'encodage et de programmation ;

— les informations étaient encodées dans des formats spécifiques en lien étroit avec les programmes qui les utilisaient. Cela rendait très difficile la réutilisation des fichiers stockés par une autre application.

Dans les années soixante, émergea l'idée de séparer les données de leurs traitements et de concevoir des logiciels spécialisés dans la gestion des données. Ces systèmes sont désignés par le nom de *systèmes de gestion de bases de données* [10], la *base de données* [11] étant le nom attribué aux collections d'informations. Les *SGBD* sont accompagnés de langages dédiés afin de recevoir les instructions de l'utilisateur et d'effectuer les tâches qui leur sont déléguées. Ces tâches sont :

— l'encodage et le *stockage physique* des données collectées ;

— l'accès aux données via des opérations prédéfinies ;

— la gestion des modifications de données ;

— le partage de données entre différentes applications.

Les *SGBD* devaient également être une solution pour sortir de la vision séquentielle imposée par le stockage physique *brut* et pour mieux exprimer les liens qui peuvent exister entre les différents enregistrements d'une base de données. Cette quête, celle de l'expressivité d'une part, et de la distanciation vis-à-vis de la représentation physique de l'autre, a guidé et guide encore aujourd'hui les concepteurs des *SGBD*. Nous présentons dans ce qui suit l'évolution des paradigmes des systèmes centrés données.

I.1.1. Paradigme navigationnel

Dans les années soixante, c'est le paradigme navigationnel qui domine et deux types de modèles de données sont proposés. Le modèle hiérarchique du système *Information Management System* [12](*IMS*) et le modèle réseau *CODALYS*.

— Le système *IMS* [13] a été créé par IBM pour la NASA dans le cadre du programme Apollo [14]. Ce système est le premier *SGBD* hiérarchique. La figure I.1 est un exemple

9. Un enregistrement est une structure de données qui correspond à une collection de champs étiquetés avec des valeurs de types hétérogènes.
10. Abrégé par *SGBD*.
11. Le terme vient des militaires américains pour désigner une collection partagée d'informations.
12. Ce système est toujours développé par IBM et est devenu un *SGBD* moderne toujours utilisé.
13. Initialement appelé *Information Control System*.
14. Il a été déployé en 1968 pour Apollo 11.

d'une organisation de données selon le modèle hiérarchique. La partie supérieure (figure I.1a) montre le schéma de données. La partie inférieure (figure I.1b) donne un exemple d'une base de données hiérarchique contenant 3 *enregistrements de données*. Chaque enregistrement de données est une collection de *segments* qui sont organisés sous forme d'un arbre hiérarchique. Chaque segment est étiqueté et possède un certain nombre de champs étiquetés pour contenir des données. Un segment peut être :

- Un segment racine : Chaque enregistrement de données contient naturellement un et un seul segment de ce type. Ce segment est le sommet de l'arbre. Dans la figure I.1, les segments `Île-de-France` et `Bretagne` sont des racines étiquetées par `Région`. Nous pouvons également remarquer la présence d'un segment racine vide, il s'agit de ce que l'on pourrait appeler une super racine relative à la base de données faisant le lien entre les différents enregistrements.

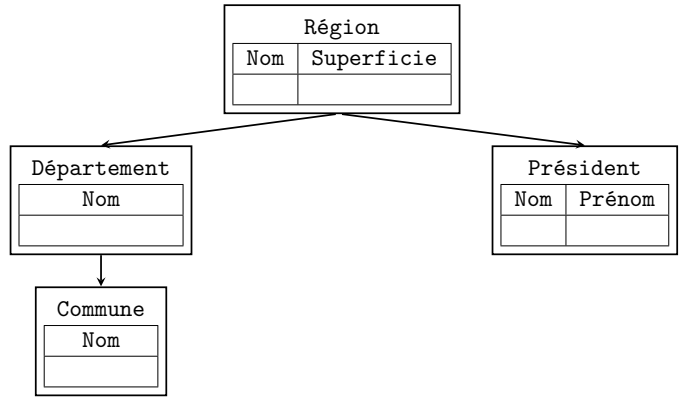
- Un segment *dépendant* d'un autre segment. Ce sont tous les segments de l'arbre sauf sa racine. Deux types de segments *dépendants* existent :
  - Un segment *père* d'un autre. Dans la figure I.1 nous avons un seul segment père : `Département`. Le segment `Région` est une racine et donc pas un segment père.
  - Un segment *fils* d'un autre. Tous les segments étiquetés par `Département`, `Président` ou `Commune` sont des segments fils dans la figure I.1.
- Un segment peut être père et fils en même temps, c'est le cas du segment `Département`.

  IMS dispose d'un langage d'accès appelé *DL/I* qualifié de *navigationnel* car il permet d'accéder aux données en explorant l'arbre. Ce système est très adapté à des systèmes d'informations très spécifiques. Nous pouvons constater que chaque relation ne peut relier que deux segments. Ce type de lien est dit $(1-1)$. Nous touchons ici à la limite d'*IMS* qui ne permet donc pas de représenter des relations $(1-N)$ c'est à dire des relations liant (1) segment à (N) segments. Par exemple, une région peut avoir plusieurs départements. Or, dans la figure I.1b, nous avons dû définir deux enregistrements pour la région `Île-de-France`, ce qui n'aurait pas était nécessaire si la relation entre `Région` et `Département` pouvait être définie comme étant de type $(1-N)$. Le modèle réseau est venu lever cette limitation.

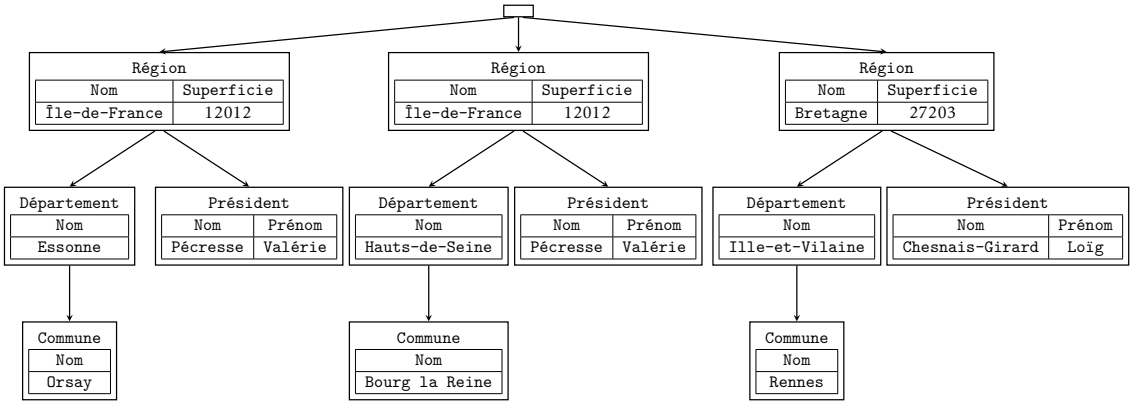
- Le modèle réseau *CODALYS* est proposé en 1969 par un consortium réuni par Charles Bachman [15]. Il vient résoudre la limitation du modèle hiérarchique en permettant de représenter les relations $(1-N)$. Là aussi le langage associé est, comme pour *IMS*, navigationnel.

Les langages navigationnels requièrent de connaître finement comment les données sont structurées, ce qui les rend difficiles à maîtriser. Il fallait trouver une représentation

15. Prix Turing 1973.



(a) *Schéma de données hiérarchique*



(b) *Base de donnée hiérarchique avec* 3 *enregistrements*

Figure I.1. *Exemple d'une base de données hiérarchique.*

plus simple à comprendre et à laquelle il était possible d'associer des langages plus simples à appréhender pour répondre aux besoins liés à l'utilisation croissante des systèmes d'informations par l'industrie non militaire et non spatiale et par le monde économique.

I.1.2. Paradigme relationnel

En 1970, Codd a défini le modèle relationnel et l'algèbre qui lui est associée [Codd, 1970]. C'est un modèle où les enregistrements sont organisés en relations homogènes. L'algèbre définit des opérateurs ensemblistes simples. Cette innovation a facilité significativement l'interaction avec les *SGBD*. Le travail de Codd [16] est considéré comme fondateur dans le domaine des bases de données. En 1974, *Sequel* [Chamberlin et Boyce, 1974], l'ancêtre de SQL, un langage proche de la langue naturelle anglaise, qui permet de manipuler des bases de données relationnelles, a été présenté par Chamberlin et Boyce. Depuis, les

16. Prix Turing 1981.

SGBD relationnels utilisant SQL sont de loin les plus répandus.

Une des caractéristiques principales du modèle relationnel est qu'il est en première forme normale (*1NF*), c'est à dire que les valeurs des attributs dans les tuples doivent toutes être atomiques [Codd, 1971]. En 1977, Makinouchi considère la norme *1NF* comme trop restrictive et propose un modèle où la valeur associée à un attribut peut être un ensemble d'ensembles [Makinouchi, 1977].

Le paradigme relationnel est au coeur de tous les paradigmes qui sont apparus après lui. Tous étendent le modèle et les opérateurs relationnels ou les utilisent.

Dans la section I.2 et la section I.3, nous présentons, respectivement, l'algèbre relationnelle et SQL, en mettant l'accent sur les éléments nécessaires à la compréhension de cette thèse.

I.1.3. Paradigmes imbriqués et paradigme objet

Dans les années quatre-vingt, de nombreux modèles de données, où les valeurs peuvent être elles-mêmes des relations, sont proposés. Bancilhon [Bancilhon *et al.*, 1982] propose une approche qui permet de traiter des relations non-normalisées en s'intéressant à l'optimisation des opérateurs unaires. [Abiteboul et Bidoit, 1986] et [Scholl *et al.*, 1989] présentent le système *Verso* qui manipule un modèle où les valeurs peuvent être des relations. Ils utilisent pour cela une représentation interne en forme d'arbre. Dans [Van Gucht et Fischer, 1988], les auteurs présentent un modèle avec des relations imbriquées à plusieurs niveaux et étendent l'algèbre relationnelle avec les opérateurs `Nest` et `Unnest` qui proposent un système permettant de restructurer les relations et de passer de *1NF* à *non-1NF* et inversement.

En 1990, Colby a présenté une approche basée sur des identifiants [Colby, 1990]. Cluet et Moerkotte [Cluet et Moerkotte, 1993] ont généralisé l'idée et ont proposé des bases de données Objet qui permettent de manipuler des objets identifiés (pointeurs) au travers de méthodes définies par l'utilisateur.

D'un point de vue théorique, ces modèles sont proches, c'est la mise en oeuvre qui diffère d'une approche à l'autre. Il est à noter également que la grande majorité des projets dont sont issus ces travaux n'ont pas eu un grand succès dans le monde réel à cause de la domination du modèle relationnel dans le monde applicatif.

En section I.4, nous présentons le langage NRA qui est un langage algébrique imbriqué et dont l'extension, NRA^e, est le langage cible de notre compilateur certifié.

I.1.4. Paradigmes semi-structurés

Dans les années quatre-vingt-dix, et avec la démocratisation du Web, des représentations de données semi-structurées voient le jour, notamment le standard XML. Ces paradigmes permettent d'intégrer des données hétérogènes. Ils reposent sur le paradigme

imbriqué. Nous nous étendrons pas sur ces paradigmes que nous jugeons inutiles à la compréhension de notre travail de recherche.

I.1.5. NoSQL

Depuis les années deux mille, avec l'apparition du Web 2.0, des applications modernes (notamment les réseaux sociaux et le e-commerce) et du *cloud computing*, les données ont pris une nouvelle dimension. Elle sont un élément central dans la révolution technologique et sociétale de ces deux dernières décennies. Cette révolution a engendré de nouveaux besoins et donc de nouveaux défis. L'apparition ou la "réhabilitation" de certains formats (modèles) de données (*JSON*, *XML*, *RDF* et *graphes*) et les langages/systèmes permettant de les manipuler (*MongoDB*, *XQuery*, *Sparql* et *Cypher*) sont une des réponses à ces défis. Ces langages et systèmes modernes sont regroupés sous le terme *NoSQL* (*Not only SQL*).

Les langages NoSQL reprennent les opérateurs de NRA et en introduisent de nouveaux selon le modèle de données auquel ils sont appliqués. Les modèles de données NoSQL sont basés sur le modèle imbriqué avec de nouveaux types de valeurs et de données. Nous avons l'intuition, d'un point de vue théorique, que tous ces langages peuvent être représentés, à un certain niveau d'abstraction, par un langage imbriqué extensible qui s'applique à un modèle imbriqué suffisamment générique pour représenter les différents modèles NoSQL.

Notre travail de thèse contribue à la première étape de la construction de DBCert : un cadre de formalisation des différents langages NoSQL et d'étude des frontières entre ces langages, notamment en définissant des traductions certifiées faisant le lien entre ces langages.

I.2. L'algèbre relationnelle (RA)

Le modèle et l'algèbre relationnels introduits dans [Codd, 1970] ont séparé la représentation logique des données de leur organisation physique. Les données sont définies selon un schéma les organisant en relations, sans se soucier de la manière dont ces données sont stockées en disque.

I.2.1. Le modèle relationnel

Le modèle relationnel est fondé sur le concept mathématique de relation. Voici un rappel de quelques notions autour des relations.

— Un *n-uple* (ou *n-uplet*) est une collection ordonnée de n éléments. Le *n-uple* dont les éléments sont, dans l'ordre, $e_1, \ldots, e_n$ est dénoté par $(e_1, \ldots, e_n)$.

— Le *produit cartésien* des ensembles $E_1,\ldots,E_n$ est l'ensemble des n-uplets $(e_1,\ldots,e_n)$ où $e_1,\ldots,e_n$ sont respectivement des éléments quelconques de $E_1,\ldots,E_n$ Le produit cartésien est dénoté par $E_1 \times \cdots \times E_n$. Par exemple, le résultat du produit cartésien $\{1,2\} \times \{3,4,5\} \times \{6\}$ est l'ensemble des triplets $\{(1,3,6),(1,4,6),(1,5,6),(2,3,6),(2,4,6),(2,5,6)\}$.

— Une relation n-aire sur les ensembles $E_1,\ldots,E_n$ est un sous-ensemble du produit cartésien $E_1 \times E_2 \times \cdots \times E_n$. Une relation sert à représenter une propriété liant des objets mathématiques. A titre d'illustration, la relation $R = \{(1,4,6),(2,3,6)\}$ sur $\{1,2\},\{3,4,5\},\{6\}$ représente les triplets (3-uplets) pour lesquels la somme des éléments est égale à 11.

— Dans le cadre des bases de données, l'*enregistrement* désigne une donnée mémorisée. Généralement, un enregistrement possède un certain nombre d'*attributs*. Chaque *attribut* représente une information élémentaire. Un attribut a un nom, un type et une valeur. Pour être plus précis, un enregistrement est *ensemble* d'attributs :

  — Les noms d'attributs doivent être distincts car ils permettent de récupérer les valeurs associées.

  — La *position* (l'ordre) des attributs n'est pas important.

— Mathématiquement, on définit un enregistrement par un *support* qui correspond à l'ensemble des noms de ses attributs et par une fonction qui associe à chacun de ces noms une valeur. Cependant, un enregistrement peut également être représenté sous la forme canonique d'un n-uple de paires nom/valeur. Lorsque un enregistrement est manipulé sous cette forme canonique, il faut prendre en compte le fait que l'ordre n'est pas important.

— Soit l'enregistrement t. On appelle $t[a_1,\ldots,a_n]$ la *restriction* de t sur l'ensemble des noms d'attributs $\{a_1,\ldots,a_n\}$. Le résultat de la restriction est l'enregistrement qui contient uniquement les attributs $\{a_1,\ldots,a_n\}$.

— Pour accéder à la valeur v associée à un nom d'attribut a dans le tuple t, on utilise une variante de la restriction qui prend un seul nom en entrée et qui produit une valeur. Elle dénotée par un point (.) : $t.a = v$.

— Un relation, en base de données relationnelle, est un ensemble d'enregistrements qui ont les mêmes noms d'attributs.

Dans le modèle relationnel, les données sont organisées en relations. Codd a utilisé des *tables* pour représenter les relations[Codd, 1970]. Pour cette raison, on parle de *tables* pour désigner les relations et on utilise toujours cette représentation graphique. Concernant les enregistrements, on a pour habitude, dans le domaine des bases de données, de les appeler *tuples* [17].

La figure I.2 montre une base de données relationnelle avec trois relations : `Livre`, `Auteur` et `LivreAuteur`. Dans ce qui suit, et en s'appuyant sur ces 3 tables données à titre

17. Néologisme issu du terme anglais *Table uple*.

Livre

lid	titre
1	Foundations of databases
2	Software foundations
3	Database management systems
4	Types and programming languages

Auteur

aid	nom	prénom
1	Abiteboul	Serge
2	Hull	Richard
3	Vianu	Victor
4	Pierce	Benjamin
5	Casinghino	Chris
6	Gaboardi	Marco
7	Ramakrishnan	Raghu
8	Gehrke	Johannes

LivreAuteur

lid	aid
1	1
1	2
1	3
2	4
2	5
2	6
3	7
3	8
4	4

Figure I.2. *Exemple d'une base de données relationnelle*

d'exemples, nous mettons en évidence certaines propriétés du modèle relationnel, tout en faisant le lien entre les concepts mathématiques et la représentation graphique.

— Chaque ligne d'une table correspond à un tuple de la relation qu'elle représente. On remarque dans la figure I.2 que les tables ont une sémantique *ensembliste*, c'est-à-dire que les tuples d'une table sont distincts. Dans cette sémantique, l'ordre des tuples n'est pas important.

— Tout comme pour les éléments d'un tuple, l'ordre des colonnes est important. Chaque colonne d'une table :

  — est étiquetée par un nom d'attribut significatif. Pour `Livre`, par exemple, il s'agit de `lid` et `titre`,

  — a un domaine bien déterminé. Dans les table `Livre`, le domaine de `lid` est `entier` et le domaine de `titre` est `texte`.

— Une table respecte un schéma de données qui est constitué à partir du nom de cette table et du tuple construit par les noms d'attributs associés à leurs domaines. Par exemple, les schémas des tables de la figure I.2 sont :

```
Livre (lid:entier,titre:texte)
Auteur (aid: entier,nom:texte,prénom:texte)
LivreAuteur (lid:entier,aid:entier)
```

— Le contenu d'une table est parfois appelé l'*extension* ou l'instance.

— Pour faire le lien entre les tables, même si ce n'est pas imposé d'un point de vue

théorique, on créé et/ou on utilise habituellement l'une des colonnes avec des valeurs distinctes comme *clé primaire*. `lid` est la clé primaire de `Livre` d'où le soulignement. Ces clés servent à créer des tables représentant des associations entre les autres tables et à identifier facilement les tuples. La `LivreAuteur` en est une illustration. Dans ce cas, on les appelle *clés étrangères*.

I.2.2. Les opérateurs relationnels

L'algèbre relationnelle recouvre les opérateurs d'union, d'intersection, de différence et le produit cartésien. Elle introduit de nouveaux opérateurs spécifiques. Nous les présentons ci-dessous en les illustrant, quand nous jugeons cela nécessaire, avec des exemples utilisant la base de données abordée précédemment dans la figure I.2.

— L'union : soient deux relations R et S de même schéma. L'union est définie par :

$$R \cup S = \{t \mid t \in R \vee t \in S\}$$

— L'intersection : soient deux relations R et S de même schéma. L'intersection est définie par :

$$R \cap S = \{t \mid t \in R \wedge t \in S\}$$

— La différence : soient deux relations R et S de même schéma. La différence est définie par :

$$R \setminus S = \{t \mid t \in R \wedge t \notin S\}$$

— Le produit cartésien : soient deux relations R et S de même schéma. Le produit cartésien est défini par :

$$R \times S = \{(r, s) \mid r \in R \wedge s \in S\}$$

— La sélection : soient une relation R, un nom d'attribut a, un nom d'attribut ou une constante x et un prédicat $\theta \in \{\leq, <, \geq, >, =, \neq\}$. La définition de la sélection est :

$$\sigma_{a\ \theta\ x}(R) = \{t \mid (t \in R) \wedge (a\ \theta\ x \text{ est vrai pour } t)\}$$

$\sigma_{\texttt{titre} = \text{Software Foundations}}(\texttt{Livre})$

lid	titre
2	Software foundations

— La projection : La projection est l'application de la restriction à tous les tuples d'une table. Le résultat est la table qui contient uniquement les attributs en indice de la projection :

$$\pi_{a_1,\ldots,a_n}(R) = \{t[a_1, \ldots, a_n] \mid t \in R\}$$

$\pi_{\texttt{titre}}(\texttt{Livre})$

titre
Foundations of databases
Software foundations
Database management systems
Types and programming languages

— Le renommage : soit un tuple t. Le résultat de $t[a/b]$ est le tuple où le nom d'attribut b a été remplacé par le nom d'attribut a. Le renommage ne modifie que la partie noms d'attribut. Il est défini par :

$$\rho_{a/\ b}(R) = \{t[a/\ b] \mid t \in R\}$$

$\rho_{\underline{\texttt{livre\_id}}/\underline{\texttt{lid}}}(\texttt{Livre})$

<u>livre_id</u>	titre
1	Foundations of databases
2	Software foundations
3	Database management systems
4	Types and programming languages

— La jointure naturelle : Soient deux relations R et S. La jointure naturelle se fait sur les noms d'attributs communs. Sa définition est donnée par :

$$R \bowtie S = \{r \circ s \circ t \mid r \circ t \in R \wedge s \circ t \in S\}$$

Dans la définition formelle ci-dessus, $\circ$ dénote la concaténation des tuples. t représente la partie commune et dénote un tuple dont les attributs sont les attributs communs à R et S. Notons que la jointure naturelle est une*composition* des relations sur les noms d'attributs communs. Notons aussi que les attributs communs ne sont conservés qu'une seule fois dans le résultat et que l'ordre des noms d'attributs n'est pas important. Si aucun attribut commun n'existe entre les deux relations, dans le cadre d'une sémantique ensembliste, la jointure naturelle devient un produit cartésien.

$\texttt{Livre} \bowtie \texttt{LivreAuteur}$

<u>lid</u>	titre	<u>aid</u>
1	Foundations of databases	1
1	Foundations of databases	2
1	Foundations of databases	3
2	Software foundations	4
2	Software foundations	5
2	Software foundations	6
3	Database management systems	7
3	Database management systems	8
4	Types and programming languages	4

— La division :

$$R/S = \{t \mid \forall\ s \in S, (t,s) \in R\}$$

R et S doivent avoir au moins un attribut commun. Le résultat de la division est l'ensemble des tuples qui, concaténés à ceux de S donnent un tuple de R.

R

a	b	c
1	1	1
1	1	2
2	1	1
3	1	1
3	2	2
4	2	2

S

c
1
2

R/S

a	b
1	1

I.3. Le langage SQL

SQL est de loin le langage centré données le plus connu et le plus utilisé. C'est un langage déclaratif regroupant trois sous-langages : le premier sert à la création des données, le deuxième à les contrôler et le dernier à les manipuler. Il a été présenté dans [Chamberlin et Boyce, 1974] [18].

Chamberlin nous explique dans [Chamberlin, 2012] pourquoi et comment est né *Sequel* devenu SQL. En 1972, Codd a organisé un colloque au centre de recherche *Watson* d'*IBM*. A cette occasion, Boyce et Chamberlin ont pu discuter avec lui sur son travail présenté en 1970 [Codd, 1970] où il introduit le modèle relationnel, l'algèbre relationnelle et le calcul relationnel. Ils étaient impressionnés par la concision qui caractérisait l'algèbre relationnelle avec la possibilité d'écrire des requêtes assez complexes en quelques lignes. Cependant, ils ont rapidement constaté qu'il fallait être habitué aux notations formelles. Ils ont ainsi travaillé sur un langage relationnel *en langue naturelle anglaise* pour qu'il soit utilisé à plus large échelle et pour contribuer à généraliser le modèle relationnel proposé par Codd.

La première version, *Sequel* [Chamberlin et Boyce, 1974], est une version *textuelle* de l'algèbre relationnelle comme l'illustre l'exemple ci-dessous.

$$\pi_{prénom}(\sigma_{nom="Pierce"}(Auteur))$$

```
select prénom
from Auteur
where nom = "Pierce";
```

Dans la lignée de Codd, *Sequel* décrit les données souhaitées mais ne dit pas comment les récupérer. Ceci conserve l'idée de séparer l'aspect logique de l'aspect physique puisque les requêtes *Sequel* se basent uniquement sur le schéma logique des données. Une version plus aboutie, *Sequel 2*, a été présentée dans [Chamberlin *et al.*, 1976]. En 1977, pour des raisons de droits commerciaux, le langage a été rebaptisé SQL.

I.3.1. Modèle de données de SQL

Le modèle de SQL reprend les bases du modèle relationnel. Les tuples sont organisés en tables. Les valeurs sont aussi atomiques. La différence réside dans la nature des tables qui dans SQL représentent des *multiensembles* [19]. Ceci a un impact important sur la sémantique des opérateurs. L'exemple présenté dans la figure I.3 illustre cette différence sur la projection. Dans la relation `LivreAuteur`, chaque tuple apparaît une seule fois. Cette relation est donc valide dans le modèle relationnel. Si on fait une projection sur

18. Boyce et Chamberlin étaient deux chercheurs chez IBM.
19. Le terme *bag* est aussi utilisé pour désigner ce type de collections.

l'attribut `LivreAuteur` de `LivreAuteur` dans l'algèbre relationnelle, on conserve une seule copie de chaque tuple. Dans SQL, toutes les occurrences sont conservées.

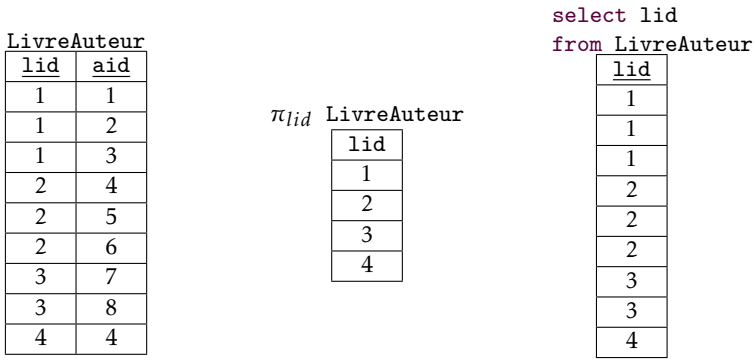


Figure I.3. *Projection ensembliste vs. Projection multiensembliste*

Nous devons rappeler à ce stade que nous considérons, dans ce travail, le fragment `select from where group by having` uniquement. Pour être plus complet, il faut dire que les tables dans SQL sont des listes [20] (séquences). Cette sémantique est imposée par la clause `order by`, qui par définition donne du sens à l'ordre des éléments. Ou même la clause `limit N` qui renvoie les `N` premiers éléments d'une table.

I.3.2. Le langage d'interrogation des données

Toute instruction SQL s'écrit comme une succession de *clauses* : `select`,`from` et `where` par exemple. Une requête (instruction, bloc) SQL doit respecter un certain nombre de règles syntaxiques comme la possibilité d'écrire tel clause avant une autre ou l'obligation de toujours associer une clause à une autre si cette dernière est utilisée.

Le langage SQL est composé de plusieurs langages dédiés à différentes tâches sur les données : Création, manipulation et contrôle. Le langage de manipulation (SQL DML) lui-même permet de réaliser un certain nombre d'opérations sur les données : ajout (`insert ...`), mise à jour (`update ...`), suppression (`delete ...`) et interrogation (`select ...`).

Dans ce travail, nous nous intéressons uniquement à l'interrogation des données. Plus précisément, nous considérons uniquement le fragment `select from where group by having`. La syntaxe complète d'une requête d'interrogation SQL est donnée en Annexe B.

Dans la figure I.4, nous montrons la syntaxe du fragment de SQL que nous traitons dans notre thèse. Les clauses s'évaluent dans un ordre bien défini :

1. `from`
2. `where`

20. L'ordre des éléments est important.

3. `group by`
4. `having`
5. `select`

select ::= SELECT [DISTINCT] [* | *expression*[[AS] *output_name*][,...]]
[FROM *table_name* [,...]]
[WHERE *condition*]
[GROUP BY (*expression* [,...])]
[HAVING *condition* [,...]]
[{UNION | INTERSECT | EXCEPT} *subquery*]

subquery ::= | *select* | *table_name*

condition ::= | TRUE | NOT *condition*
| *condition* AND *condition* | *condition* OR *condition*
| EXISTS *subquery* | *expresion compare_operator expresion*
| *expresion compare_operator* ANY (*subquery*)
| *expresion compare_operator* ALL (*subquery*)
| *expression*[[AS] *output_name*][,...] IN *subquery*

Figure I.4. *Syntaxe du fragment* `select from where group by having` *de SQL*

Exemples de requêtes

Nous présentons ci-dessous quelques exemples sur lesquels nous nous appuyons pour donner l'intuition sur le fonctionnement des différentes clauses que nous considérons. Nous reprenons l'instance de la base de données présentée dans la figure I.2.

Dans le premier exemple, nous voulons récupérer les titres des livres écrits par `Pierce` et nous savons que l'identifiant de `Pierce` est 4.

```
select Livre.titre as titre
from Livre, LivreAuteur
where Livre.lid =LivreAuteur.lid and  LivreAuteur.aid = 4;
```

titre
Types and programming languages

Dans la clause `select`, on énonce les attributs qu'on souhaite récupérer : `Livre.titre` et on peut renommer un attribut avec le mot-clé `as` (titre). L'opérateur `T.col` dit que

l'attribut `col` se trouve dans la table `T` et permet de lever les ambiguïtés lorsque plusieurs tables ont des attributs avec le même nom. Dans la clause `from`, on liste tous les noms des tables qu'on interroge dans la requête. Enfin, dans la clause `where`, on définit les conditions de filtrage. Dans l'exemple, la présence des deux tables dans la clause Sqlfrom engendre un produit cartésien et la condition `Livre.lid = LivreAuteur.lid` permet de réaliser une jointure naturelle puisque il s'agit du seul attribut commun.

Dans le second exemple, nous voulons associer à chaque auteur le nombre de livres qu'il a rédigés.

```
select Auteur.nom as auteur,count(lid) as n_livres
from Auteur, LivreAuteur
where Auteur.aid =LivreAuteur.aid
group by auteur;
```

Résultat sans le `group by` et le `count`

auteur	lid
Abiteboul	1
Hull	1
Vianu	1
Pierce	2
Pierce	4
Casinghino	2
Gaboardi	2
Ramakrishnan	3
Gehrke	3

Résultat de la requête

auteur	n_livres
Abiteboul	1
Hull	1
Vianu	1
Pierce	2
Casinghino	1
Gaboardi	1
Ramakrishnan	1
Gehrke	1

Dans cet exemple, nous utilisons l'agrégat `count` qui compte le nombre de n-uplets. Dans SQL, l'utilisation d'un agrégat est associée à l'utilisation de la clause `group by` qui permet de préciser sur quel les n-uplets sont groupées, *i.e.*, que les lignes du résultat sont découpées en groupes selon les valeurs des attributs groupants.

En ajoutant la clause `having` à la requête précédente, nous pouvons, par exemple, obtenir les noms des auteurs qui ont écrit au moins deux livres :

```
select Auteur.nom as auteur,count(lid) as n_livres
from Auteur, LivreAuteur
where Auteur.aid =LivreAuteur.aid
group by auteur
having n_livres >= 2;
```

auteur	n_livres
Pierce	2

Enfin, une version différente de cette requête nous permet de récupérer les noms des auteurs uniquement et qui montre que l'agrégat peut se situer ailleurs que dans la clause `select` :

```
select Auteur.nom as auteur
from Auteur, LivreAuteur
where Auteur.aid =LivreAuteur.aid
group by auteur
having count(lid) >= 2;
```

auteur
Pierce

I.3.3. Spécificités de SQL

SQL a connu plusieurs extensions qui l'ont éloigné de sa sémantique originale basée sur l'algèbre relationnelle. Les deux aspects les plus subtiles de la sémantique de SQL concernent les valeurs nulles et les requêtes corrélées.

I.3.3.1. Les valeurs nulles

Dans SQL, les valeurs nulles sont basées sur une logique trivaluée [21]. Soient deux tables :

— `R : [{a:1, b: 10},{a:null, b:20}]`

— `S : [{a:null}]`

Prenons une requête adaptée de [Guagliardo et Libkin, 2017] appliquée à une base qui contient les deux tables `R` et `S` :

```
select R.a as a
from R
where R.a not in (select S.a from S);
```

Intuitivement, nous pourrions penser que l'évaluation de cette requête est `[{a:1}]`, ce qui est incorrecte. Le résultat de la requête `select S.a from S` est le contenu de `S : [{a:null}]`. Ainsi, évaluer le prédicat `not in` revient à évaluer `1 <> null` pour la premier n-uplet de `R` et `null <> null` pour la second. Dans les deux cas le résultat est `unknown`, ces deux n-uplets ne sont pas considérés dans le résultat de la requête. Le résultat correcte de la requête est une relation vide : `[]`.

I.3.3.2. Les requêtes corrélées

Contrairement à d'autres langages centrés données, SQL permet de *corréler* les requêtes imbriquées, c'est-à-dire que les noms d'attributs spécifiques à une requêtes peuvent apparaître dans d'autres requêtes, et lorsque c'est le cas, ces noms d'attributs doivent être évalués dans leur *contexte* d'origine.

21. La logique trivaluée admet 3 valeurs : `true`, `false` et `unknown`.

Considérons l'exemple de la figure I.5, la seule différence entre les deux requêtes est que dans `Q1` l'opérateur `sum` contient `a2`, alors que dans `Q2`, il contient `a1`. A première vue, les expressions (`sum(1+0*a1) = 2`) et (`sum(1+0*a2) = 2`) peuvent toutes les deux être simplifiées par (`sum(1) = 2`). Dans ce cas, `Q1` et `Q2` sont équivalentes. Cela est incorrect. En effet, le nom d'attribut a un impact sur le résultat puisque le contexte d'évaluation en dépend.

T1

a1	b1
1	1
1	2
2	3
3	1
3	2
3	3

T2

a2	b2
7	7
7	8

Q1

```
select a1
from T1
group by a1
having exists (select a2
               from T2
               group by a2
               having sum(1+0*a2) = 2);
```

Résultat

a1
1
2
3

Q2

```
select a1
from T1
group by a1
having exists (select a2
               from T2
               group by a2
               having sum(1+0*a1) = 2);
```

Résultat

a1
1

Figure I.5. *Example de requêtes corrélées*

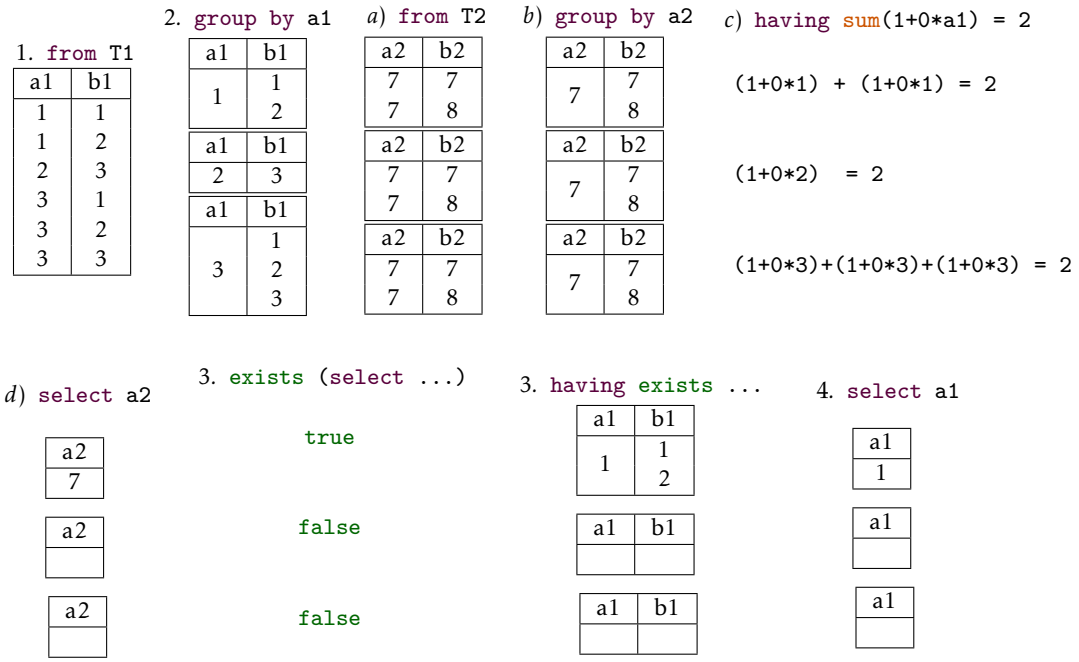
La requête `Q1` n'est pas corrélée, c'est-à-dire qu'on peut obtenir le résultat de la sous-requête indépendemment de la requête elle-même. Calculons le résultat de la sous-requête. Le résultat de `select a2 from t2` est `[{a2:7},{a2:7}]`. Ensuite, nous groupons par rapport aux valeurs de `a2` (`group by a2`). Pour la valeur 7, nous avons deux lignes. Pour cette valeur, le résultat de `having sum(1+0*a2) = 2` est vrai : `sum(1+0*a2) = (1+0*7) + (1+0*7) = 2`. Donc la valeur 7 est renvoyée dans le résultat. Ainsi, le résultat de la sous-requête est `[{a2:7}]`. Revenons à présent à la requête, nous pouvons omettre la clause `having` car la formule `exists` est toujours vraie, puisque la résultat de la sous-requête n'est jamais vide. Le résultat de `Q1` est donc `[{a1:1},{a2:2},{a2:3}]`.

La requête `Q2` est corrélée. En effet, le nom d'attribut `a1` qui apparaît dans la sous-requête `select a2 from T2 group by a2 having sum(1+0*a1) = 2` vient du contexte

externe (`select a1 from T1 group by a1`)[22]. Analysons les étapes de l'évaluation :

1. `from T1` : on récupère le contenu de la table `T1`.
2. `group by a1` : on sépare les lignes en différents groupes selon les valeurs de `a1`.
3. `having (exists ...)` : pour chacun des groupes, on garde uniquement les lignes pour lesquels la condition est vraie. Pour savoir si la condition est vraie, on doit évaluer la sous-requête pour chacun des groupes :
   a) `from T2`
   b) `group by a2`
   c) `having sum(1+0*a1) = 2` : pour savoir si la condition est vraie, nous appliquons l'agrégat `sum` au groupe de `a1`. C'est ici que se situe la corrélation, puisqu'on va chercher la valeur de `a1` dans le contexte externe. Autrement dit, la variable `a1` est libre dans la sous-requête et et liée dans la requête.
4. `select a1` : la dernière étape est la projection sur `a1`. Chaque groupe est transformé en une ligne dans cette étape. Si une expression avec agrégat est présente à coté de l'attribut groupant, alors on applique l'agrégat aux lignes du groupe. Sinon, si seul l'attribut groupant est présent, comme dans cet exemple, alors on élimine les duplicats.

Voici un schéma représentant les étapes décrites ci-dessus :



Le résultat final est `[{a1:1}]` car la seule valeur qui apparaît uniquement deux fois dans la colonne `a1` de `T1` est la valeur `1` et donc c'est la seule valeur pour laquelle le

22. L'attribut `a1` est libre dans la sous-requête et liée dans la requête.

`having sum(1+0*a1) = 2` est vrai.

Ces deux aspects produisent des *bugs* et des divergences entre les différentes implémentations. Parmi les travaux importants qui ont apporté des réponses à cette limitation, [Benzaken et Contejean, 2019] propose une sémantique mécanisée en Coq de SQL et une extension de l'algèbre relationnelle qui prend en compte tous ces aspects et qui capture une fragment important de la sémantique de SQL. Ce projet est à la base de notre travail de recherche. A notre connaissance, au moment de la rédaction de cette thèse, c'est la sémantique formelle mécanisée de SQL la plus aboutie. Nous présentons plus longuement cette sémantique dans la section III.1.

I.4. L'algèbre imbriquée (NRA)

L'algèbre relationnelle imbriquée ou NRA pour *Nested Relational Algebra* a été introduite dans [Cluet et Moerkotte, 1993]. Cette algèbre, qui est une extension de l'algèbre de GOM [Kemper et Moerkotte, 1990], a été conçue pour l'optimisation de requêtes imbriquées de langages de données objets. Afin d'exploiter les optimisations de requêtes relationnelles existantes dans le cadre de données objets, NRA étend les opérateurs relationnels pour qu'ils s'appliquent à des données objets et introduit de nouveaux opérateurs spécifiques.

Plusieurs différences sont à noter entre l'algèbre relationnelle et NRA, en voici quelques unes utiles à la compréhension de la traduction certifiée présentée dans la Deuxième partie :

1. Là où le résultat d'une requête relationnelle est toujours une collection *plate* de tuples, le résultat d'une requête NRA, lui, peut être n'importe quelle valeur, simple ou plus ou moins complexe (imbriquée).
2. La deuxième différence, en lien avec la première, est que dans NRA, une requête n'est pas toujours une opération algébrique et peut exprimer une expression.
3. Dans les requêtes relationnelles, l'imbrication et l'éventuelle corrélation entre requêtes se situe au niveau de la clause de sélection alors que dans NRA, elle peut se situer dans n'importe quelle clause.
4. Les relations sont des *multiensembles*, on peut donc avoir plusieurs fois le même tuple dans une collection et la sémantique tient compte des duplications.

I.4.1. Modèle de données

Le modèle de données de NRA est proche de celui de GOM [Kemper et Moerkotte, 1990] et de O_2 [Deux et al., 1990]. Le modèle contient :

— des classes avec notion d'héritage instanciables en objets avec un identifiant unique.

Table Livre

Titre	Auteurs
Foundations of databases	Nom: Abiteboul, Hull, Vianu; Prénom: Serge, Richard, Victor
Software foundations	Nom: Pierce, Casinghino, Gaboardi; Prénom: Benjamin, Chris, Marco
Database management systems	Nom: Ramakrishnan, Gehrke; Prénom: Raghu, Johannes
Types and programming languages	Nom: Pierce; Prénom: Benjamin

Figure I.6. *Exemple d'une relation dans le modèle NRA*

Ces classes sont manipulables avec des méthodes,

— des valeurs simples ou complexes manipulables par des opérateurs spécifiques.

La figure I.6 montre une représentation dans le modèle imbriqué de l'exemple de base de données de la figure I.2 [23]. Dans le paradigme imbriqué, on a besoin d'une seule table pour représenter les données, là où dans le paradigme plat, on a utilisé 3 tables. Ainsi, les associations de type $(1-N)$ peuvent être représentées par une valeur qui est elle même une relation [24]. Cependant, nous remarquons que l'auteur Pierce Benjamin, qui a écrit 2 livres, apparaît 2 fois dans la table, là où dans le modèle plat, il apparaît une seule fois dans la table `Auteur` et c'est sa clé qui apparaît 2 fois dans la table `LivreAuteur`.

I.4.2. Les opérateurs de NRA

L'algèbre NRA étend l'algèbre GOM [Kemper et Moerkotte, 1990]. Les opérateurs sont polymorphes. Ils prennent n'importe quels types d'expressions en arguments, y compris des requêtes. NRA est orientée optimisation, c'est pour cette raison que l'algèbre n'est pas minimale [25]. Ci-dessous les opérateurs que comprend l'algèbre NRA. Comme pour les deux sections précédentes, nous illustrons la présentation des opérateurs, lorsque cela est nécessaire, en utilisant l'exemple de base de données de la figure I.6.

— Les opérateurs : $\cup$, $\cap$ et $\setminus$ ont une sémantique multiensembliste et sont donc

23. Qui a servi à illustrer les concepts de l'algèbre relationnelle en section I.2.
24. Plus précisément, c'est une collection de tuples.
25. Certains opérateurs peuvent être définis à partir d'autres.

définis sur la multiplicité (le nombre d'occurrences). Pour l'union, on prend la somme des multiplicités pour chacun des éléments de l'un des deux multiensembles. L'intersection s'obtient avec la multiplicité minimale des éléments appartenant aux deux ensembles. Enfin, la différence correspond au multiensemble construit en prenant n(t) fois chaque tuple t dans le premier multiensemble avec n égal au nombre d'apparitions de t dans le premier multiensemble moins le nombre d'apparitions dans le second.

— La sélection : $\sigma_p(e) = \{x \mid x \in e, p(x)\}$. La sélection s'applique uniquement à des expressions qui dénotent des collections. Contrairement à l'algèbre relationnelle, ici p dénote une fonction quelconque et non plus seulement des prédicats.

— La jointure : $e_1 \bowtie_p e_2 = \{x \circ y \mid x \in e_1, y \in e_2, p(x,y)\}$. La jointure s'applique à deux expressions qui dénotent des collections et ne se limite pas à l'égalité mais s'applique à un prédicat binaire quelconque p. Le symbole $\circ$ représente ici la concaténation des enregistrements, ce qui signifie que cet opérateur jointure n'est défini que pour des collections d'enregistrements.

— Le `map` : $\chi_{e_2}(e_1) = \{e_2(x) \mid x \in e_1\}$. Cet opérateur est bien connu en programmation fonctionnelle [26]. La projection et le renommage sont des cas particuliers du `map`. Cet opérateur est très important dans NRA puisqu'il permet d'appliquer un opérateur quelconque (algébrique ou pas) à tous les éléments d'une collection (de tuples ou de tout autre type de données). Ce qui suit est un exemple de projection.

$\chi_{\{\texttt{Titre}:x.\texttt{Titre}\}}(\texttt{Livre}[x])$

Titre
Foundations of databases
Software foundations
Database management systems
Types and programming languages

`Livre`$[x]$ dit que x est la variable qui désigne les éléments de `Livre` lors du parcours séquentiel de tous ses tuples. Avec l'indice {`Titre` : x.`Titre`}, on construit, pour chaque élément x de `Livre`, un nouveau tuple {`Titre` : x.`Titre`}, où x.`Titre` est la valeur associée au nom d'attribut `Titre` dans x. Il est à noter que χ s'applique à une collection d'éléments, quelque que soit le type de ces éléments, et non seulement pour les tuples. De même, le résultat est une collection dont les éléments sont de n'importe quel type. C'est pour cette raison qu'on fabrique un tuple pour faire une projection.

— Le *grouping* unaire : $\Gamma_{g;A\theta;f}(e) = \{y.A \circ [g : G] \mid y \in e, G = f(\{x \mid x \in e, x.A\theta y.A\})\}$:

  — Le symbole $\circ$ dénote la concaténation des tuples.

  — Le symbole . représente ici la restriction d'un tuple sur un ensemble de noms d'attributs.

  — L'évaluation de l'expression e doit être une collection.

26. La fonction `map f l` applique la fonction `f` à tous les éléments de la liste `l` et retourne la liste obtenue.

— g est le nom de l'attribut qui va contenir les groupes.

— $A\theta$ est le critère de regroupement : A est un ensemble de noms d'attributs et θ un prédicat.

— f est une fonction à appliquer à chaque groupe, dans le résultat.

Un cas particulier de cet opérateur est $\Gamma_{g;A=;proj}(e)$. On appelle cet opérateur le `nest` ou parfois le `group by`. C'est ce cas particulier que nous utilisons dans le compilateur que nous proposons. Il s'agit simplement de regrouper selon les valeurs (égales) d'un ensemble d'attributs. La figure I.7 en illustre un exemple. Dans la requête $\Gamma_{\texttt{Auteurs};[\texttt{Titre}]=;proj}(\texttt{Livre})$, $proj$ dénote la projection sur tous les noms d'attributs excepté les groupantes.

Livre

titre	Nom	Prénom
Foundations of db	Abiteboul	Serge
Foundations of db	Hull	Richard
Foundations of db	Vianu	Victor
Software foundations	Pierce	Benjamin
Software foundations	Casinghino	Chris
Software foundations	Gaboardi	Marco
Database management sys.	Ramakrishnan	Raghu
Database management sys.	Gehrke	Johannes
Types and programming languages	Pierce	Benjamin

$\Gamma_{\texttt{Auteurs};[\texttt{Titre}]=;proj}(\texttt{Livre})$

Titre	Auteurs
Foundations of db	Nom: Abiteboul, Prénom: Serge Nom: Hull, Prénom: Richard Nom: Vianu, Prénom: Victor
Software foundations	Nom: Pierce, Prénom: Benjamin Nom: Casinghino, Prénom: Chris Nom: Gaboardi, Prénom: Marco
Database management sys.	Nom: Ramakrishnan, Prénom: Raghu Nom: Gehrke, Prénom: Johannes
Types and programming languages	Nom: Pierce, Prénom: Benjamin

Figure I.7. *Exemple de l'application d'un `group by` (NRA)*

— Le `grouping` binaire, le `unnest`, le `flatten` et le `max` : nous n'utilisons pas ces opérateurs dans ce travail, nous nous contentons donc de les citer.

I.5. Comparaison entre RA, SQL et NRA

Pour la compréhension de la suite de ce document, nous avons pensé qu'il était intéressant de faire un petit état des lieux des similarités et des différences entre les trois langages que nous venons de présenter : RA, SQL et NRA. La table I.1 résume cette comparaison.

Les valeurs dans l'algèbre relationnelle sont atomiques [27]. Pour SQL, malgré la richesse des types proposés, les valeurs restent atomiques. Enfin, concernant NRA, les valeurs sont naturellement complexes puisque ce langage est destiné aux données objets.

Pour la sémantique des relations, elle est ensembliste pour RA et multiensembliste pour SQL [28] et NRA.

Enfin, là où dans RA et SQL on distingue bien entre les requêtes (expressions algébriques), les formules et les expressions, que ce soit au niveau syntaxique ou au niveau de l'évaluation, dans NRA, tous les opérateurs sont polymorphes, ce qui a pour conséquence l'unification des expressions algébriques, des formules et des expressions quelconques dans le même élément syntaxique global qu'on appelle expression NRA ou requête NRA. De même pour l'évaluation, pour RA et SQL, chaque catégorie de briques syntaxiques est évaluée de manière spécifique à celle-ci, et dans NRA, une évaluation globale est définie.

	RA	SQL	NRA
Valeurs	Atomiques	Atomiques	Complexes
Collections	Ensemble	Multiensemble	Multiensemble
Evaluation	Requêtes	Requêtes	Expressions

Table I.1. *Tableau comparatif entre RA, SQL et NRA*

I.6. Conclusion

Dans ce chapitre, nous avons introduit les langages centrés données à travers une présentation historique. Le modèle et l'algèbre relationnels fondent ou inspirent tous les autres modèles et langages centrés donnés. Le langage SQL a une sémantique fondée sur l'algèbre relationnelle, mais les relations sont multiensemblistes et le langage est plus expressif, avec notamment les valeurs nulles la possibilité de corréler les requêtes. Le langage NRA est une extension de l'algèbre relationnelle avec un modèle qui permet l'imbrication des relations et avec de nouveaux opérateurs. Le langage source et cible de la traduction présentée dans cette thèse sont équivalents, respectivement, à SQL et NRA. La traduction est mécanisée et certifiée en Coq en s'appuyant sur les méthodes formelles,

27. Les types complexes comme les collections ou les tuples ne sont pas autorisés.
28. Si l'on considère la clause `order by` ou `limit`, les tables deviennent des séquences et l'ordre devient singificatif. Dans cette thèse, on se limite au fragment `select from where group by having`.

plus précisément, les méthodes de preuve formelle. Le chapitre suivant est dédié à la présentation de ces méthodes.

Chapitre

II

Compilation certifiée en Coq

La traduction que nous présentons dans cette thèse est formalisée et certifiée correcte avec l'assistant de preuve Coq. Les travaux effectués sous Coq et la méthodologie suivie pour réaliser notre travail peuvent être classés dans la catégorie des preuves formelles, elle-même faisant partie des méthodes formelles, l'un des deux cadres contextuels de ce travail.

II.1 Méthodes formelles **28**
II.2 Assistants de preuve **28**
II.2.1 Coq 29
II.3 Compilation certifiée correcte **30**
II.3.1 Définitions 30
II.3.1.1 Compilation 30
II.3.1.2 Encodage et décodage 30
II.3.1.3 Mécanisation d'un langage 30
II.3.1.4 Compilateur algébrique 31
II.3.2 Correction d'un compilateur algébrique 31
II.3.2.1 Encodage et Décodage isomorphiques 31
II.3.2.2 Encodage homomorphique 32
II.3.2.3 Décodage homomorphique 32
II.3.3 Compilation certifiée en Coq d'un langage jouet 32
II.3.3.1 Le langage `ToyExp` 33
II.3.3.2 Instanciations de `ToyExp` 33
II.3.3.3 Définition et certification de `ToyExpCert` 33
II.3.3.4 Instanciation du compilateur `ToyExpCert` 34
II.4 Conclusion **35**

II.1. Méthodes formelles

Très tôt, dès 1949, Maurice Wilkes, un des premiers programmeurs, a fait le constat que la possible présence de *bugs* est intrinsèque à tout programme informatique et que l'effort d'élimination de ces *bugs* prendra une part importante dans le processus de création de programmes *justes* qui réalisent *correctement* ce que l'on attend d'eux.

Dans l'industrie du logiciel, les différentes phases du développement sont accompagnées de tests pour garantir que le programme *réalise* le comportement initialement *spécifié*. Mais ces garanties deviennent insuffisantes lorsque la criticité du logiciel augmente. En effet, comme l'explique si bien Dijkstra, « Le test peut être utilisé pour montrer la présence de *bugs*, mais en aucun cas leur absence ».

D'autres méthodes formelles apportent plus de garanties sur la *correction* d'un logiciel. Ces méthodes, qui trouvent leurs racines dans l'*informatique fondamentale*, sont utilisées pour vérifier certains logiciels critiques, dans l'avionique ou dans l'industrie du *hardware* par exemple, mais les méthodes formelles ont gagné leurs lettres de noblesse dans la recherche scientifique. Le principe de ces méthodes est de démontrer, par calcul et déduction logique, qu'une ou plusieurs propriétés sont vraies pour toutes les exécutions possibles d'un programme.

Ces méthodes nécessitent beaucoup de rigueur et sont irréalisables sans une assistance de l'ordinateur et une automatisation.

Dans sa leçon inaugurale au Collège de France [Leroy, 2019], Xavier Leroy classifie les outils utilisés pour formellement vérifier des logiciels en 4 catégories reflétant différentes méthodes formelles :

— L'analyseur statique permet d'inférer des propriétés simples sur les variables d'un programme.

— Le vérificateur par modèle permet de vérifier des propriétés de la logique temporelle en explorant les états atteignables pendant l'exécution du programme.

— Le vérificateur déductif, permet de vérifier qu'un programme est complètement conforme à sa *spécification* en l'annotant avec des assertions logiques.

— L'assistant de preuve permet de mener des raisonnements mathématiques trop complexes pour être automatisés. La démonstration est construite en interaction avec l'utilisateur, mais c'est l'assistant qui se charge de sa vérification.

II.2. Assistants de preuve

Les assistants de preuve offrent un cadre de raisonnement (semi-)automatique sur des spécifications. Ils permettent de définir des spécifications de systèmes avec la possibilité de les réaliser (implémenter), puis de vérifier des propriétés en utilisant ces spécifications. Ces outils sont fondés sur différentes logiques. Dans le cadre de la spécification des

langages formels, la logique d'ordre supérieur offre une riche expressivité pour formaliser et démontrer des propriétés sur une large variété de langages. Les outils les plus matures basés sur cette logique sont l'instance Isabelle/HOL d'Isabelle [1] et Coq.

Beaucoup de travaux de recherche sur les langages formels ont privilégié Coq. La logique intuitionniste et les types dépendants sont parfois mis en avant pour expliquer ce choix, mais à notre avis, ce qui est décisif dans le choix d'utiliser Coq est son mécanisme d'extraction qui permet d'extraire un système *opérationnel* et *exécutable* à partir des spécifications et des réalisations. Dans le cadre de la mécanisation d'une sémantique ou de la traduction d'un langage, l'intérêt d'un tel mécanisme d'extraction est évident : obtenir *gratuitement* une version exécutable du système formalisé.

II.2.1. Coq

Coq est un assistant de preuve basé sur une logique d'ordre supérieur : le calcul des constructions inductives. Il permet l'expression de spécifications et le développement de programmes cohérents avec leur spécifications [Bertot et Castéran, 2010].

Coq permet de développer différents types de projets. Il peut par exemple servir à formaliser et à prouver des théorèmes mathématiques comme le théorème de Feit-Thompson [Gonthier *et al.*, 2013]. Il offre également la possibilité de mécaniser [2] une sémantique comme JSCert [Bodin *et al.*, 2014], qui est une mécanisation d'un fragment de JavaScript en Coq, ou de certifier un compilateur avec le projet de référence CompCert [Leroy *et al.*, 2016], qui est un compilateur certifié du langage *C*. Dans cette thèse, nous utilisons Coq pour spécifier et définir un compilateur certifié.

L'approche générale [3] pour vérifier certaines propriétés d'un logiciel est de définir une *spécification formelle* de ce logiciel puis de définir une *instanciation* (ou *réalisation*) de cette spécification. Cela est possible avec le langage Gallina permettant d'écrire des termes pouvant définir des fonctions, des programmes ou des théorèmes. Enfin, les théorèmes correspondants aux propriétés à prouver sont énoncés en Gallina et leur démonstration est construite avec le langage de tactiques Ltac.

Coq comprend également un mécanisme d'extraction de code qui permet, par exemple, d'extraire un compilateur certifié écrit en OCaml. La traduction que nous proposons fait partie du compilateur certifié DBCert qui traduit du SQL en code JavaScript. Ce compilateur a été obtenu en utilisant le mécanisme d'extraction.

1. Isabelle est un outil générique qui permet de travailler dans différentes logique.
2. Voir II.3.1.3. Mécanisation d'un langage.
3. Voir l'exemple jouet dans II.3.3. Compilation certifiée en Coq d'un langage jouet.

II.3. Compilation certifiée correcte

II.3.1. Définitions

II.3.1.1. Compilation

Pour décrire de la manière la plus simple un *compilateur*, nous dirons que que c'est un programme qui *traduit*/*compile* un langage *source* en un langage *cible*. Cette traduction est appelée *compilation*. Il est important ici de comprendre que le compilateur prend en entrée de la *syntaxe* et qu'il produit également de la syntaxe. Par exemple, tout traducteur d'une langue naturelle vers une autre peut être considéré comme un compilateur.

II.3.1.2. Encodage et décodage

Dans ce cadre, l'*encodage* est la transcription d'une donnée, d'un état ou d'un environnement du modèle source dans le modèle cible. Par opposition, le *décodage* est la transcription qui permet de passer du modèle cible au modèle source. Ici l'entrée et la sortie se placent au niveau sémantique. Pour illustrer ces concepts, nous prendrons l'exemple d'un langage source dont le domaine d'évaluation correspond aux booléens et d'un langage cible évalué dans l'ensemble des entiers. Voici un exemple d'encodage `enc` et de décodage `dec` :

$$\begin{aligned}&\texttt{enc}(\texttt{false}) = 0\\&\texttt{enc}(\texttt{true}) = 1\\&\texttt{dec}(0) = \texttt{false}\\&\texttt{dec}(1) = \texttt{true}\end{aligned}$$

Nous pouvons remarquer que la définition de `dec` n'est pas complète. En effet, il manque les cas de tous les entiers strictement supérieurs à 1. Pour ces valeurs, nous pouvons les transcrire en `true` comme le font la majorité des langages de programmation. Nous pouvons aussi décider que le décodage pour ces valeurs n'est pas défini. En effet, contrairement à l'encodage, le décodage peut être partiel dans ce cadre de travail, puisque, comme expliqué dans la sous-section II.3.2, le décodage est appliqué à des entrées obtenues soit après encodage soit après compilation, on peut donc se contenter de définir un décodage pour les sorties que l'encodage produit.

II.3.1.3. Mécanisation d'un langage

Aujourd'hui, la majorité des travaux conséquents sur la vérification des langages de programmation s'appuient sur des outils pour automatiser une partie importante du processus de vérification. La communauté qui utilise les assistants de preuve utilise le terme de mécanisation pour décrire ce processus qui comprend :

— La spécification du langage et la définition de sa sémantique.

— La définition des propriétés qu'on veut vérifier.

— La démonstration de la validité de ces propriétés.

II.3.1.4. Compilateur algébrique

Les méthodes algébriques de traduction sont souvent utilisées pour construire un compilateur. On parle dans ce cas de traduction algébrique. Ces méthodes sont utilisées lorsque on veut traduire une logique en une autre logique ou une *sémantique* en une autre sémantique. Les approches algébriques peuvent plus ou moins diverger d'un champ de recherche à un autre.

Un compilateur permet de traduire un langage source en un langage cible. Dans l'approche que nous présentons ici et qui est utilisée dans ce travail, ces deux langages sont algébriques.

II.3.2. Correction d'un compilateur algébrique

Lorsque un compilateur assure que la traduction préserve *correctement* la sémantique, on dit que ce compilateur est *correct*. Comme expliqué dans [Janssen, 1998], la notion générale de correction formelle d'un compilateur algébrique est toujours définie par la *commutativité* d'un diagramme de *simulation*. Le diagramme de simulation regroupe les langages (les syntaxes) et les sémantiques sources et cibles. Ils constituent les sommets du diagramme. On y trouve également les différentes traductions (compilation pour la syntaxe et encodage pour la sémantique) et les interprétations source et cible. Ce sont les transitions du diagramme. Toujours dans [Janssen, 1998], on trouve une comparaison intéressante des différentes visions et approches vis-à-vis de la correction d'un compilateur algébrique, chacune reflétée par un diagramme de simulation spécifique. Nous présentons dans ce qui suit les 3 visions applicables au compilateur que nous avons construit.

II.3.2.1. Encodage et Décodage isomorphiques

La figure II.1 correspond à la conception de la correction d'un compilateur selon [Mosses, 1980]. Mosses propose que l'encodage et le décodage soient des isomorphismes (des bijections), c'est-à-dire :

$$\forall\ x, \texttt{enc}(\texttt{dec}(x)) = x \wedge \forall\ y, \texttt{dec}(\texttt{enc}(y)) = y$$

Une vision encore plus restrictive a été proposée dans [Polak, 1981]. Celle-ci propose carrément que l'encodage et le décodage soient la fonction *identité*. Autrement dit, que le langage source et le langage cible aient le même domaine ou univers.

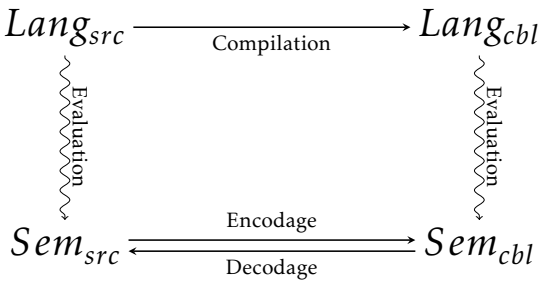


Figure II.1. *Correction de compilation selon [Mosses, 1980]*

II.3.2.2. Encodage homomorphique

L'approche la plus répandue est celle, proposée dans [Thatcher *et al.*, 1980], où l'encodage est un homomorphisme (Injection). Mais dans [Janssen, 1998], des exemples de langages pour lesquels cette approche est problématique sont donnés.

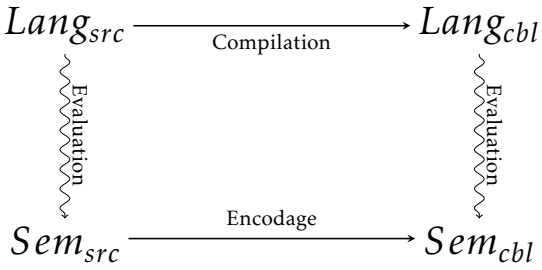


Figure II.2. *Correction de compilation selon [Thatcher* et al.*, 1980]*

II.3.2.3. Décodage homomorphique

Cette approche [Morris, 1973] est la duale de la précédente. Ici c'est le décodage qui est injectif. Cette approche est celle que privilégie Janssen. Car il préconise que la vérification de l'équivalence des sémantiques doit se faire coté source. Il est vrai que depuis le début des années 2000, c'est cette approche qui est le plus souvent adoptée [4].

II.3.3. Compilation certifiée en Coq d'un langage jouet

Afin de concrétiser les différentes notions exposées dans ce chapitre, nous proposons d'appliquer la méthodologie qui a permis d'élaborer le compilateur certifié présenté dans cette thèse sur un langage minimal. Dans cet exemple, nous cherchons à compiler un

4. Par exemple dans CompCert.

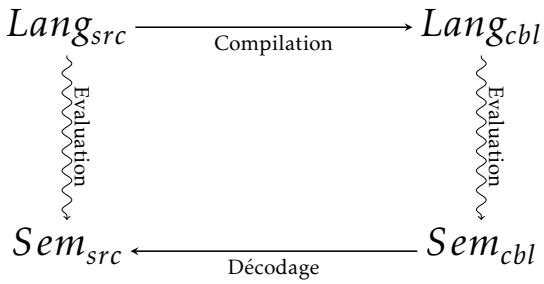


Figure II.3. *Correction de compilation selon [Morris, 1973]*

langage à un seul opérateur binaire correspondant à l'opérateur $\wedge$ de l'algèbre de bool en un langage à un seul opérateur binaire correspondant à l'opérateur $*$ de l'arithmétique. Les booléens sont le domaine source et les entiers le domaine cible.

Nous pouvons facilement remarquer la similarité des deux langages. Il est intéressant de définir une *classe* de langages les regroupant, puis l'instancier cette classe pour la faire correspondre à l'un ou l'autre de nos deux langages.

II.3.3.1. Le langage `ToyExp`

Le langage `ToyExp` est le langage des expressions construites à partir de valeurs dans un domaine et d'un certain nombre d'opérateurs binaires. La figure II.4 présente la définition d'un tel langage en Coq. On spécifie un domaine générique `domain`, des opérateurs binaires génériques `binary_op` et on suppose l'existence d'une fonction d'évaluation pour ces opérateurs binaires génériques. C'est une sorte de trous ou de boites vides qui sont à instancier pour obtenir différentes implémentations de cette classe de langages. Cette spécification est utilisée pour écrire des définitions et des propriétés qui sont valables quelle que soit l'instanciation.

II.3.3.2. Instanciations de `ToyExp`

Pour réaliser notre compilateur jouet, nous avons besoin de deux langages qu'on obtient en instanciant `ToyExp`. La figure II.5 montre ces deux instances. Nous définissons un type d'opérateurs binaire avec un seul opérateur `star`. nous définissons également sa fonction d'évaluation pour chacun des deux langages, puis nous construisons nos deux instances.

II.3.3.3. Définition et certification de `ToyExpCert`

Nous appelons notre compilateur jouet `ToyExpCert`. Au lieu de nous contenter d'un compilateur des booléens vers les entiers, nous définissons un compilateur générique qui

```
Module ToyExp.

  Class Specification : Type :=
    mk_C {
        value : Type;
        binary_op : Type;
        eval_binary_op : binary_op → value → value → value
      }.

  Section Def.
    Context {Spec : Specification}.

    Inductive expression :=
    | binexp : binary_op → expression → expression → expression
    | val : value → expression.

    Fixpoint eval_expression e :=
      match e with
      | val v ⇒ v
      | binexp bop e1 e2 ⇒
        eval_binary_op bop (eval_expression e1) (eval_expression e2)
      end.
  End Def.

End ToyExp.
```

Figure II.4. *Spécification/Définition du langage ToyExp*

fonctionne pour toute la classe de langages ToyExp et que nous allons ensuite instancier pour réaliser notre objectif.

Nous supposons qu'il existe une instance du langage source SourceSpec et une instance du langage cible TargetSpec. Nous supposons également détenir une fonction d'encodage et une fonction de traduction des opérateurs génériques correctes. Cette spécification nous permet de définir une fonction de traduction translation et de prouver qu'elle préserve la sémantique en utilisant l'hypothèse de correction de la traduction des opérateurs sources. La figure II.6 montre la formalisation de ToyExpCert.

II.3.3.4. Instanciation du compilateur ToyExpCert

Une fois la traduction générique définie, il ne reste plus qu'à définir une instance de la spécification de la traduction. En proposant un encodage correct et une traduction correcte de l'opérateur star, nous obtenons un compilateur parfaitement fonctionnel, comme l'illustre la figure II.7.

```
Section Instances.
  Import Expression.

  Inductive binary_operator :=  star.

  Definition eval_binary_operator_bool s a b :=
    match s with  star ⇒ andb a b end.

  Definition eval_binary_operator_nat s a b :=
    match s with  star ⇒ a * b end.

  Instance BoolAnd :Specification :=
    mk_C bool binary_operator eval_binary_operator_bool.

  Instance NatMult :Specification :=
    mk_C nat binary_operator eval_binary_operator_nat.

  Definition bool_to_nat a :=
    match a with | true ⇒ 1 | false ⇒ 0 end.
End Instances.
```

Figure II.5. *Instanciations du langage ToyExp*

II.4. Conclusion

Ce chapitre donne une vue générale sur les méthodes dites formelles, notamment les méthodes de preuve formelle. Il présente un exemple très simple de formalisation d'un langage et d'une traduction certifiée sous l'assistant de preuve Coq. Cet exemple donne l'intuition sur la méthodologie que nous avons appliquée. Dans le chapitre suivant, qui fait le lien entre les deux premiers chapitres, nous présentons deux projets de formalisation en Coq de langages centrés données, le projet SqlCert et le projet DBCert pour comprendre de manière plus détaillée la méthodologie appliquée à des langages centrés données tout à fait réalistes et pour permettre de comprendre la contribution de cette thèse qui est une mise en lien entre ces deux projets, en proposant une traduction certifiée correct de l'un des langages du premier projet, vers l'un de ceux du second.

```
Module Compilation.
  Import ToyExp.
  Class ToyExpCert
        `{SourceSpec : ToyExp.Specification}
        `{TargetSpec : ToyExp.Specification} : Type :=
    mk_C {
        encode : (@value SourceSpec) → (@value TargetSpec);
        translate_binop :
          (@binary_op SourceSpec) → (@binary_op TargetSpec);
        translate_binop_correct :
          ∀b v1 v2 ,
            eval_binary_op (translate_binop b) (encode v1) (encode v2) =
            encode (eval_binary_op b v1 v2);
      }.

  Section Def.

    Context {SSpec : ToyExp.Specification}.
    Context {TSpec : ToyExp.Specification}.
    Context {Spec : @ToyExpCert.Specification SSpec TSpec}.

    Fixpoint translation (e:@expression SSpec):@expression TSpec :=
      match e with
      | val v ⇒ val  (encode v)
      | binexp bop e1 e2 ⇒
        binexp (translate_binop bop) (translation e1) (translation e2)
      end.

    Lemma correctness :
      ∀e, eval_expression (translation e) = encode (eval_expression e).
    Proof.
      induction e;[|reflexivity].
      -  simpl;rewrite IHe1,IHe2;apply translate_binop_correct.
    Qed.
  End Def.
End Compilation.
```

Figure II.6. *Définition et certification de `ToyExpCert`*

```
Section BoolAnd2NatMult.

  Import Expression.

  Definition encode_bool_to_nat a :=
    match a with | true ⇒ 1 | false ⇒ 0 end.

  Definition translate_binary_operator (bop:binary_operator) :=
    match bop with
    | star ⇒ star
    end.

  Lemma translate_binary_operator_correct:
    ∀b v1 v2 ,
      (@eval_binary_op NatMult)
        (translate_binary_operator b)
        (encode_bool_to_nat v1) (encode_bool_to_nat v2) =
      encode_bool_to_nat (@eval_binary_op BoolAnd b v1 v2).
  Proof.
    destruct b;simpl.
    destruct v1;destruct v2;apply eq_refl.
  Qed.

  Instance BoolAnd2NatMult :@Compilation.Specification BoolAnd NatMult
    := @Compilation.mk_C
        BoolAnd NatMult encode_bool_to_nat translate_binary_operator
        translate_binary_operator_correct.

End BoolAnd2NatMult.
```

Figure II.7. *Instanciation du compilateur* `ToyExpCert`

Chapitre

III

Formalisation de SQL et de NRA

Nous présentons dans ce chapitre le langage source et le langage cible du compilateur certifié proposé dans le chapitre IV de cette thèse, respectivement, SQL_{Alg} et NRA^e. SQL_{Alg} [Benzaken et Contejean, 2019] est une extension de l'algèbre relationnelle proposée dans le cadre du projet Datacert. Ce langage capture la sémantique du fragment `select from where group by having` de SQL. Quant à NRA^e [Auerbach *et al.*, 2017b], c'est une mécanisation d'une version de NRA qui permet la modification de l'environnement au niveau syntaxique. Ce langage fait partie d'une série de langages proposés dans la chaîne de compilation certifiée Q*Cert [Auerbach *et al.*, 2017d].

III.1 SqlCert **40**
III.1.1 Présentation de SqlCert 40
III.1.2 SQL_{Alg} : Une extension de RA pour SQL 41
III.1.2.1 Représentation des données 42
III.1.2.2 Syntaxe 43
III.1.2.3 Exemples de requêtes 46
III.1.2.4 Les environnements de SQL_{Alg} 46
III.1.2.5 Sémantique 49
Sémantique des expressions sans agrégats 50
Sémantique des expressions avec agrégats 51
Sémantique des formules 52
Sémantique des requêtes 53
III.1.2.6 Preuve de concept : Une instance de la spécification . . 54
III.2 Q*Cert **57**
III.2.1 Présentation de Q*Cert 57
III.2.2 NRA^e 57
III.2.2.1 Représentation des données 57
III.2.2.2 Syntaxe de NRA^e 58
III.2.2.3 Sémantique de NRA^e 60
III.3 Conclusion **62**

III.1. SqlCert

Dans le cadre du projet Datacert, V. Benzaken et É. Contejean proposent SQL_{Coq}, une sémantique mécanisée en Coq du fragment `select from where group by having` de SQL [Benzaken et Contejean, 2019]. Un des résultats importants issus de ce projet est la proposition d'une extension de l'algèbre relationnelle adaptée à SQL [Benzaken *et al.*, 2018]. Cette algèbre, appelée SQL_{Alg}, est également mécanisée en Coq. La démonstration de l'équivalence de sémantique entre SQL_{Coq} et SQL_{Alg} est effectuée et validée sous Coq. Nous présentons dans cette section SQL_{Coq} et SQL_{Alg} en mettant plus l'accent sur ce dernier et en explicitant les éléments que nous avons jugés utiles à la compréhension du chapitre IV.

III.1.1. Présentation de SqlCert

SqlCert est un travail de formalisation en Coq du fragment `select from where group by having` de SQL. SqlCert inclut deux langages : SQL_{Coq} et SQL_{Alg}. SQL_{Coq} est une mécanisation de ce fragment de SQL en Coq. Dans SQL_{Coq}, les clauses `where` et `group by having` sont obligatoires pour uniformiser les requêtes. Le renommage est également obligatoire, pour la même raison. SQL_{Alg} est une algèbre relationnelle étendue inspirée de celle présentée dans [Garcia-Molina *et al.*, 2009]. Une traduction, dans les deux sens, est proposée pour ces deux langages. Ces traductions sont accompagnées d'une preuve en Coq de la préservation de sémantique. Ce travail a permis de démontrer sous Coq l'équivalence entre SQL [1] (SQL_{Coq}) et de cette extension de l'algèbre relationnelle (SQL_{Alg}). Ces deux langages sont multiensemblistes comme le langage SQL [2]. Ils prennent en compte les valeurs nulles, les agrégats et les quantificateurs. Ils permettent d'exprimer les requêtes imbriquées, notamment les requêtes corrélées.

$$
\begin{aligned}
query ::= \;& | \; table \\
& | \; query \; (\texttt{union} \mid \texttt{intersect} \mid \texttt{except}) \; query \\
& | \; \texttt{select} \; (\texttt{*} \mid \overline{e^a \; \texttt{as} \; attribute}) \\
& \quad \texttt{from} \; \overline{(query(\overline{attribute \; \texttt{as} \; attribute}))} \\
& \quad \texttt{where} \; formula \\
& \quad \texttt{group by} \; \overline{e^f} \\
& \quad \texttt{having} \; formula
\end{aligned}
$$

Figure III.1. *Syntaxe de SQL_{Coq}*

A titre d'indication, la syntaxe de SQL_{Coq} est présentée dans la figure III.1. Nous n'abordons pas la sémantique de SQL_{Coq} dans cette thèse pour éviter toute confusion, portant l'attention sur SQL_{Alg} qui est le langage source de la traduction certifiée que nous avons définie.

1. Le fragment `select from where group by having`.
2. Les clauses `order by` et `limit`, qui imposent que les tables soient des listes, ne sont pas considérées.

Ci-dessous deux exemples de traduction en SQL_{Coq} de requêtes SQL :

```
-- sql
select titre
from livre;
```

⟶

```
-- sqlcoq
select  titre as c_titre
from (livre (lid as c0_lid,titre as
    c0_titre))
where true
group by None
having true;
```

```
select nom as auteur,
       count(lid) as n_livres
from Auteur, LivreAuteur
where Auteur.aid = LivreAuteur.aid
group by auteur
having n_livres >= 2;
```

↓

```
select c0_nom as auteur,
       count(c1_lid) as n_livres
from (Auteur (aid as c0_aid,
              nom as c0_nom,
              prenom as c0_prenom)
     ),
     (LivreAuteur
              (lid as c1_lid,
              aid as c1_aid))
where c0_aid = c1_aid
group by c0_nom
having count(c1_lid) >= 2;
```

III.1.2. SQL_{Alg} : Une extension de RA pour SQL

Dans le langage SQL_{Alg}, les relations dénotent des multiensembles de tuples plats. En plus des opérateurs relationnels classiques, SQL_{Alg} admet un nouvel opérateur γ qui permet d'exprimer la sémantique des clauses `group by` et `having`. Les sous-requêtes peuvent être corrélées, c'est-à-dire que dans une sous-requête, peut apparaître un nom d'attribut relatif au résultat d'une autre requête, par exemple :

$$\sigma_{b>3}(\pi_{(\texttt{a as b})}\texttt{A})$$

Pour une table `A` avec un seul attribut `a`, la requête σ est corrélée à la requête π car l'attribut b qui apparaît dans la formule est celui que produit le résultat de la sous-requête π. Ce comportement est géré à travers les environnements de SQL_{Alg} et la façon dont ces derniers sont manipulés durant l'évaluation. Nous estimons que la réflexion fine et complète de cet aspect est fondamental et et qu'elle est décisive dans ce travail. Ci-dessous, nous donnons plus de détails sur l'encodage des environnements et la traduction de leur manipulation.

III.1.2.1. Représentation des données

Le modèle de données est générique. Il peut être instancié de différentes manières [3]. Cette approche permet une certaine flexibilité et une adaptation aux différentes implémentations de SQL disponibles. Elle permet également de se situer au bon niveau d'abstraction pour la formalisation et pour les preuves.

Ce modèle de données est défini par les éléments suivants :

1. Un ensemble d'étiquettes génériques pour représenter les noms d'attributs. Cet ensemble est dénoté par $\mathcal{L}abel$.
2. On distingue quatre types de données dans ce modèle :
   - Des valeurs primitives génériques. Leur ensemble est dénoté par $\mathcal{V}alue$.
   - Des tuples abstraits dont l'ensemble est dénoté par $\mathcal{T}uple$. Un tuple est défini par un ensemble d'étiquettes (s) et une fonction totale qui renvoie pour chaque étiquette de s une valeur associée et pour tout autre étiquette une valeur par défaut. Pour tout tuple t :
     - la fonction $\ell(t)$ renvoie l'ensemble d'étiquettes de t ;
     - la fonction $(t \cdot et)$ applique la fonction totale de t à l'étiquette et.

     Ces tuples sont également représentables par une forme canonique qui est une liste d'associations nom d'attribut/valeur.
   - Des multiensembles de tuples qu'on appelle $\mathcal{B}ag_{\mathcal{T}}$.
   - Des booléens abstraits : $\mathcal{A}Bool$. Le fait d'avoir des booléens abstraits permet de supporter la logique trivaluée pour la prise en compte des valeurs nulles, mais également, la logique classique. Ce choix est toujours dans l'optique de la généricité du modèle.

SQL_{Alg} est formalisé en Coq avec la même approche que le langage `ToyExp` présenté dans II.3.3.1, avec une formalisation paramétrée par une spécification générique. La figure III.2 montre la partie de la spécification [4] qui paramètre SQL_{Alg}. La spécification est appelée `Rcd`. En plus des 4 types de domaines cités précédemment, on remarque la

3. Voir II.3.3.1. Le langage `ToyExp`.
4. La spécification complète est présentée en Annexe D.

présence de fonctions de comparaison et quelques propriétés, notamment la propriété `tuple_eq` qui dit que deux tuples sont égaux si et seulement s'ils ont le même ensemble de noms d'attributs et si la valeur associée à chacun des ces noms d'attributs est la même dans les deux tuples.

```
    Record Rcd : Type :=
mk_R
  {
    attribute : Type; (* Label *)
    value : Type;    (* Value *)
    tuple : Type; (* Tuple *)
    B : Bool.Rcd;       (* ABool *)
    (** Comparaisons *)
    (* Oset.Rcd : Definition d'un ordre et d'une egalite syntaxique *)
    OAtt: Oset.Rcd attribute;
    OVal: Oset.Rcd value;
    (* Oeset.Rcd : Definition d'un ordre et d'une equivalence *)
    OTuple: Oeset.Rcd tuple;
    (** Fonctions associees aux tuples *)
    (* Fset.set A : Ensemble de noms d'attributs *)
    labels : tuple → Fset.set A;       (* ℓ(_) *)
    dot : tuple → attribute → value;       (* _·_ *)
    (** Proprietes *)
    tuple_eq : ∀t1 t2,
        (t1 ≡_t t2) ↔
        (labels t1 =S= labels t2
         ∧ ∀a, a ∈ (labels t1) → dot t1 a = dot t2 a);
    (* ... *)
  }.
```

Figure III.2. *Modèle générique paramétrant la formalisation Coq de SQL_{Alg}*

III.1.2.2. Syntaxe

Les expressions

Les expressions sont partagées par SQL_{Coq} et SQL_{Alg}. Ceci simplifie les traductions et les preuves dans SqlCert. Les expressions dénotent des valeurs. Elles sont construites à partir de valeurs ou des noms d'attributs, d'abord sans agrégats e^f, et ensuite avec e^a. Cette distinction entre les deux types d'expressions est utile à la construction, à la vérification de la bonne formation et à l'évaluation des expressions. La syntaxe des expressions est présentée dans la figure III.3. Il est à noter que les expressions sont paramétrées par des symboles de fonction et des symboles d'agrégats génériques [5] dénotés respectivement par

5. Voir II.3.3.1. Le langage `ToyExp`.

$\mathcal{F}unctionSymbol$ et $\mathcal{A}ggregateSymbol$. La figure III.4 montre la partie de la spécification de SqlCert qui concerne les symboles de fonctions et les agrégats. Là aussi, on dispose de fonctions de comparaison. On dispose également de fonctions d'interprétations.

$$
\begin{array}{rll}
e^f ::= & v & v \in \mathcal{V}alue \\
& \mid a & a \in \mathcal{L}abel \\
& \mid \mathtt{fn}(\overline{e^f}) & \mathtt{fn} \in \mathcal{F}unctionSymbol \\
e^a ::= & e^f & \\
& \mid \mathtt{ag}(e^f) & \mathtt{ag} \in \mathcal{A}ggregateSymbol \\
& \mid \mathtt{fn}(\overline{e^a}) & \mathtt{fn} \in \mathcal{F}unctionSymbol
\end{array}
$$

Figure III.3. *Syntaxe des expressions SQL_{Alg}*

```
Record Rcd : Type :=
  mk_R
    {
      (* ... *)
      symbol : Type;     (* FunctionSymbol *)
      aggregate : Type;     (* AggregateSymbol *)
      (** Comparaisons *)
      OSymb : Oset.Rcd symbol;
      OAgg : Oset.Rcd aggregate;
      (** Interpretations *)
      interp_symbol : symbol → list value → value; (* [[ ]]_f *)
      interp_aggregate : aggregate → list value → value; (* [[ ]]_a *)
      (* ... *)
    }.
```

Figure III.4. *Fonctions et agrégats génériques paramétrant les expressions SQL_{Alg}*

Les formules

Les formules sont également partagées par SQL_{Coq} et SQL_{Alg}. La figure III.5 présente la syntaxe des formules. En plus de la conjonction, la disjonction et la négation, la formalisation est dotée d'un ensemble de symboles de prédicat n-aires génériques dénoté par $\mathcal{P}redicateSymbol$. Il est possible d'associer un quantificateur universel `all` ou existentiel `any` aux prédicats. Une formule peut également être construite à partir du connecteur `exists` appliqué à une requête qui vérifie si la requête renvoie une collection non vide. Enfin, une formule peut être un connecteur `in` associé à une liste d'expressions nommées, qui dénote un tuple [6] et une requête, et qui teste la présence de ce tuple dans le résultat

6. Une liste de paire d'expressions/noms d'attributs.

de cette requête. La figure III.6 montre la partie de la spécification de SqlCert qui définit les prédicats génériques, une fonction pour les comparer et une interprétation.

$$
\begin{array}{rl}
formula ::= & \mid \texttt{true} \\
& \mid \texttt{not}\ formula \\
& \mid formula\ (\texttt{and} \mid \texttt{or})\ formula \\
& \mid \mathfrak{pr}(\overline{e^a}) \qquad\qquad \mathfrak{pr} \in \mathcal{P}redicateSymbol \\
& \mid \mathfrak{pr}(\overline{e^a}, (\texttt{all} \mid \texttt{any})\ query) \qquad\qquad \mathfrak{pr} \in \mathcal{P}redicateSymbol \\
& \mid \overline{e^a\ \texttt{as}\ attribute}\ \texttt{in}\ query \\
& \mid \texttt{exists}\ query
\end{array}
$$

Figure III.5. *Syntaxe des formules de SQL_{Alg}*

```
Record Rcd : Type :=
  mk_R
    {
      (* ... *)
      predicate : Type;     (* PredicateSymbol *)
      (** Comparaison *)
      OPred : Oset.Rcd predicate;
      (** Interpretations *)
      interp_predicate : predicate → list value → Bool.b B; (* [[ ]]_p *)
      (* ... *)
    }.
```

Figure III.6. *Prédicats génériques paramétrant les formules SQL_{Alg}*

Les requêtes

$$
\begin{array}{rl}
query ::= & \mid table \\
& \mid query\ (\texttt{union} \mid \texttt{intersect} \mid \texttt{except})\ query \\
& \mid query \bowtie query \\
& \mid \pi_{(\overline{e^a\ \texttt{as}\ attribute})}(query) \\
& \mid \sigma_{formula}(query) \\
& \mid \gamma_{(\overline{e^a\ \texttt{as}\ attribute}, \overline{e^f}, formula)}(query)
\end{array}
$$

Figure III.7. *Syntaxe des requêtes SQL_{Alg}*

SQL_{Alg} étend l'algèbre relationnelle en introduisant un nouvel opérateur γ permettant d'exprimer les clauses `group by` et `having`. Sa syntaxe est pésentée dans la figure III.7.

III.1.2.3. Exemples de requêtes

Soit une instance de base de données avec deux relations `S : (a:integer)` et `R : (a:integer, b: integer)`.

Nous reprenons la requête que nous avons utilisé pour donner l'intuition sur le comportement de SQL concernant les valeurs nulles :

```
select R.a as a from R where R.a not in (select S.a from S);
```

La traduction [7] de cette requête en SQL_{Alg} est :

$$\pi_{(\texttt{a as a})}(\sigma_{(\texttt{not}((\texttt{a as a})\texttt{in}(\pi_{(\texttt{a as a})}S)))}R)$$

Reprenons à présent la requête `Q2` de l'exemple explique la corrélation de requêtes dans SQL :

```
create table T1 (a1 integer, b1 integer);
create table T2 (a2 integer,b2 integer);

select a1
from T1
group by a1
having exists (select a2
               from T2
               group by a2
               having sum(1+0*a1) = 2);
```

Voici une requête équivalente en SQL_{Alg} :

$$\gamma_{(\texttt{a1 as a1,a1,exists}(\gamma_{(\texttt{a2 as a2,a2,sum}(1+0*\texttt{a1})=2)})\texttt{T2})}\texttt{T1}$$

III.1.2.4. Les environnements de SQL_{Alg}

Description des environnements de SQL_{Alg}

Un environnement d'évaluation est une pile. Les éléments de cette pile, appelés *tranches*, sont composés d'un ensemble de noms d'attributs, d'une liste d'expressions groupantes optionnelle et d'un paquet de tuples. La figure III.8 est une vue graphique d'un exemple d'environnement de SQL_{Alg}. La tranche se trouvant au sommet de la pile possède deux noms d'attributs h et i, le symbole – signifie que la tranche n'a pas de liste d'expressions groupantes. Cette tranche possède également un paquet contient uniquement un tuple $[\{h : 1, i : 2\}]$. La tranche la plus interne de la pile a trois noms d'attribut x, y et i, l'attribut z est une expression groupante et le paquet possède quatre

7. La traduction présentée ici est simplifiée par réécriture.

tuples. Ces deux exemples illustrent les deux formes possibles de tranches dans un environnement d'évaluation.

Notons que ensembles de noms d'attributs et les listes d'expressions groupantes des tranches d'un environnement sont une information purement statique. Cependant, cette information est indispensable à l'évaluation car elle permet d'indexer les environnements.

La première forme de tranches est sans liste d' expressions groupantes et comporte un paquet contenant uniquement un tuple. La deuxième forme possède des expressions groupantes et son paquet comprend potentiellement plusieurs tuples. Chacune de ces deux formes correspond à une des deux situations possibles durant l'évaluation. Nous y revenons un peu plus loin dans cette section.

La présence des noms d'attributs et des expressions groupantes dans les tranches est utile à l'évaluation. Pour les noms d'attributs, ils nous permettent de vérifier rapidement si un nom d'attribut appartient à ceux du paquet courant. Quant aux expressions groupantes, elles nous permettent de définir les critères de regroupement.

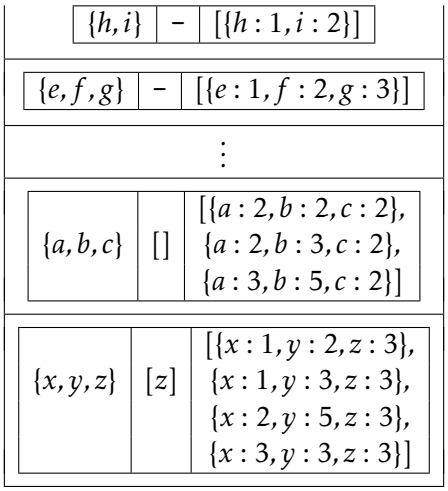


Figure III.8. *Exemple d'un environnement de SQL_{Alg}*

Bonne formation des environnements

Dans SQL_{Alg}, une notion de bonne formation est associée aux environnements d'évaluation. La bonne formation est conservée par construction durant l'évaluation d'une requête bien formée. En effet, la définition de la bonne formation étant récursive, cette préservation est assurée par descente récursive et est également préservée par la fonction d'évaluation. Cette notion de bonne formation est très utile pour les preuves Coq car elle inclut des propriétés nécessaires dans le processus de démonstration. L'environnement présenté dans la figure III.8 est bien formé.

La définition de la notion de bonne formation des environnements est :

1. L'environnement vide est bien formé.
2. L'ajout d'une tranche *bien formée par rapport à un environnement* dans cet environnement forme un environnement bien formé.

Une tranche est bien formée si :

1. Les noms d'attributs d'une tranche correspondent aux noms d'attributs des tuples de son paquet.
2. Le paquet de tuples d'une tranche n'est jamais vide.
3. Dans une tranche sans expressions groupantes, le paquet contient un et un seul tuple.
4. Dans une tranche avec des expressions groupantes, les noms d'attributs qui font partie des expressions groupantes sont associés à la même valeur dans tous les tuples du paquet de la tranche.

La notion de bonne formation d'une tranche par rapport à un environnement est la suivante :

1. La tranche est bien formée
2. Les ensembles des noms d'attributs de cette tranche et de toutes les tranches d'un environnement sont distincts.

Gestion des environnements dans SQL_{Alg}

Le langage SQL permet de construire des requêtes imbriquées. Elles peuvent être corrélées ou pas. On retrouve naturellement cette propriété dans SQL_{Alg}. Elle est reflétée dans la gestion des environnements durant l'évaluation d'une requête. Pour mieux comprendre la gestion des environnements, nous nous intéressons à la requête SQL suivante :

```
select a from A where a in (select b from B)
```

Dans cette requête, on a une imbrication due à l'opérateur `in` qui s'applique à une sous-requête. La traduction en SQL_{Alg} de cette requête après quelques simplifications donne :

$$\sigma_{(\texttt{a in } \pi_{(\texttt{b as b})}\texttt{B})}(\pi_{(\texttt{a as a})}\texttt{A})$$

Les équation qui suivent explicitent quelques étapes de l'évaluation de la requête précédente :

$$[\![\sigma_{(\texttt{a in } \pi_{(\texttt{b as b})}\texttt{B})}(\pi_{(\texttt{a as a})}\texttt{A})]\!]^{\text{q}}_{<i,[]>} = \{\!|t \in [\![\pi_{(\texttt{a as a})}\texttt{A}]\!]^{\text{q}}_{<i,[]>} \mid [\![(\texttt{a in } \pi_{(\texttt{b as b})}\texttt{B})]\!]^{\text{b}}_{<i,[(\ell(t),-,[t])]>} = \top |\!\}$$

La fonction d'évaluation est paramétrée par une instance de la base de données i et un environnement vide []. Dans la deuxième équation, on peut voir que pour évaluer l'opérateur σ, on commence par évaluer la sous-requête $[\![\pi_{(\texttt{a as a})}\texttt{A}]\!]^{\text{q}}_{<i,[]>}$. Ensuite, chaque tuple du résultat obtenu est ajouté sous forme d'une tranche à l'environnement courant : $\ell(t), -, [t]$, et on évalue la formule sous ces nouveaux environnements. Dans le résultat final, on ne conserve que les tuples pour lesquels l'évaluation de la formule est vraie : $[\![(\texttt{a in } \pi_{(\texttt{b as b})}\texttt{B})]\!]^{\text{b}}_{<[(\ell(t),-,[t])],=>} \top$.

On s'intéresse à présent à l'évaluation de la formule `in`. Voici sa définition :

$$[\![(\texttt{a in } \pi_{(\texttt{b as b})}\texttt{A})]\!]^{\text{b}}_{<i,[(\ell(t),-,[t])]>} \quad = \quad [\![a]\!]^{\text{a}}_{[(\ell(t),-,[t])]} \in [\![\pi_{(\texttt{b as b})}\texttt{B}]\!]^{\text{q}}_{<i,[(\ell(t),-,[t])]>}$$

On constate que la formule est vraie lorsque l'évaluation de l'expression $[\![a]\!]^{\text{a}}_{([\ell(t),-,[t])]}$ appartient à $[\![\pi_{(\texttt{b as b})}\texttt{B}]\!]^{\text{q}}_{<i,[(\ell(t),-,[t])]>}$, c'est-à-dire qu'elle appartient à l'évaluation de la sous-requête. On observe également que l'expression et la sous-requête s'évaluent sous un environnement non vide : $[(\ell(t), -, [t])]$.

Enfin, concernant l'évaluation de l'expression : $[\![a]\!]^{\text{a}}_{[(\ell(t),-,[t])]}$, qui, dans cet exemple, est un nom d'attribut, on l'évalue en explorant les tranches de l'environnement afin de trouver celle où ce nom d'attribut apparaît. Retenons ici que durant l'évaluation une ou plusieurs tranches peuvent être éliminées de l'environnement. Ici, a appartient aux attributs de $t : \ell(t)$. Donc, l'évaluation est simplement la valeur associée à l'attribut a dans l'un des tuples du paquet $[t]$ donc dans t.

III.1.2.5. Sémantique

Avant de présenter la sémantique formelle des expressions, des formules et des requêtes, nous introduisons les fonctions d'interprétation de chacun de ces éléments syntaxiques :

— Expressions simples : $[\![e^f]\!]^{\text{f}}_{\mathcal{E}} = v$, où $\mathcal{E}$ est un environnement. Les expressions simples dénotent des valeurs génériques ($v \in \mathcal{V}alue$).

— Expressions complexes : $[\![e^a]\!]^{\text{a}}_{\mathcal{E}} = v$, où $\mathcal{E}$ est un environnement. Les expressions complexes dénotent des valeurs génériques ($v \in \mathcal{V}alue$).

— Formules : $[\![f]\!]^{\text{b}}_{<i,\mathcal{E}>} = b$, où $\mathcal{E}$ est un environnement. Les expressions complexes dénotent des valeurs de la logique trivaluée ($b \in \mathcal{A}Bool$).

— Requêtes : $[\![q]\!]^{\text{q}}_{<i,\mathcal{E}>} = m$, où i est une instance de la base de données et $\mathcal{E}$ un environnement. Les expressions complexes dénotent des multiensembles de tuples ($m \in \mathcal{B}ag_{\mathcal{T}}$).

Sémantique des expressions sans agrégats

La fonction d'évaluation des expressions sans agrégats est dénotée par $[\![\ ]\!]^{\mathrm{f}}$. Nous présentons dans ce qui suit la sémantique des expressions sans agrégats [8] :

— L'évaluation d'une constante c est la valeur qu'elle représente :

$$[\![c]\!]^{\mathrm{f}}_{\mathcal{E}} = c \qquad \text{si } c \in \mathcal{V}alue$$

— Pour évaluer un attribut a, il faut aller trouver la valeur qui lui correspond dans l'environnement :

1. Pour un environnement vide, on renvoie une valeur par défaut dénotée par `default` :

$$[\![a]\!]^{\mathrm{f}}_{[]} = \texttt{default} \qquad \text{si } a \in \mathcal{L}abel$$

Un environnement d'évaluation d'une expression bien formé n'est jamais vide. Cette équation nous permet uniquement d'avoir une fonction d'évaluation totale.

2. Lorsque l'environnement a au moins une tranche, on regarde si a appartient aux noms d'attributs de la tranche. Si ce n'est pas le cas, on élimine la tranche courante. Si c'est le cas, cela veut dire que dans tous les tuples du paquet courant, une valeur [9] est associée à cet attribut. Enfin, on renvoie la valeur associée à cet attribut dans l'un des tuples du paquet de la tranche courante $(t.a)$:

$$\begin{aligned}
[\![a]\!]^{\mathrm{f}}_{(A,G,T)::\mathcal{E}} &= [\![a]\!]^{\mathrm{f}}_{\mathcal{E}} && \text{si } a \notin A \\
[\![a]\!]^{\mathrm{f}}_{(A,G,t::T)::\mathcal{E}} &= t.\mathrm{a} && \text{si } a \in A \\
[\![a]\!]^{\mathrm{f}}_{(A,G,[])::\mathcal{E}} &= \texttt{default} && \text{si } a \in A
\end{aligned}$$

La situation décrite par la dernière équation ne se produit jamais durant l'évaluation d'une requête avec un environnement bien formé et n'est défini que pour que la fonction d'évaluation soit totale.

— L'évaluation d'un symbole de fonction $\mathtt{fn}$ appliqué à une liste d'expressions sans agrégat $(\overline{e^f})$ est obtenue par simple descente récursive, puis par l'application de l'interprétation du symbole de fonction dénotée $[\![\ ]\!]_f$:

$$[\![\mathtt{fn}(\overline{e^f})]\!]^{\mathrm{f}}_{\mathcal{E}} = [\![\mathtt{fn}]\!]_f(\overline{[\![e^f]\!]^{\mathrm{f}}_{\mathcal{E}}}) \qquad \text{si } \mathtt{fn} \in \mathcal{F}unctionSymbol$$

8. On les qualifiera parfois d'expressions simples.
9. Cette valeur est la même pour tous les tuples du paquet si l'expression et l'environnement sont bien formés.

Sémantique des expressions avec agrégats

La fonction d'évaluation des expressions avec agrégats (expressions complexes) est dénotée par $[\![\ ]\!]^{\mathrm{a}}$. Nous présentons dans ce qui suit la sémantique des expressions complexes :

— Si l'expression est une expression sans agrégat (e^f), on applique l'évaluation des expressions sans agrégats $[\![\ ]\!]^{\mathrm{f}}$:

$$[\![e^f]\!]^{\mathrm{a}}_{\mathcal{E}} = [\![e^f]\!]^{\mathrm{f}}_{\mathcal{E}}$$

— S'il s'agit d'une liste d'expressions complexes ($\overline{e^a}$) portée par un symbole de fonction $\mathfrak{fn}$, on fait une descente récursive et on applique l'interprétation du symbole de fonction :

$$[\![\mathfrak{fn}(\overline{e^a})]\!]^{\mathrm{a}}_{\mathcal{E}} = [\![\mathfrak{fn}]\!]_f(\overline{[\![e^a]\!]^{\mathrm{a}}_{\mathcal{E}}}) \qquad \text{si } \mathfrak{fn} \in \mathcal{F}unctionSymbol$$

— Enfin, lorsqu'il s'agit d'une expression simple (e^f) portée par un symbole d'agrégat, on commence par trouver le bon environnement d'évaluation. La figure III.9 détaille la sémantique de la fonction $\mathbb{F}_e(\mathcal{E}, e^f)$ qui permet de déterminer quel est ce bon environnement. La figure ne montre que les cas pouvant se produire lorsque l'environnement est bien formé. Les autres cas, définis uniquement pour que cette fonction soit totale, n'y sont pas montrés. Les fonctions $\mathbb{B}_u(G, e^f)$, $\mathbb{S}_e(A, \mathcal{E}, e^f)$ et $\mathbb{F}_e(\mathcal{E}, e^f)$ renvoient une valeur de vérité.

  — La fonction $\mathbb{B}_u(G, e^f)$ vérifie si une expression simple e^f est *construite* à partir d'une liste d'expressions groupantes.

  — La fonction $\mathbb{S}_e(A, \mathcal{E}, e^f)$ permet de vérifier si un environnement $\mathcal{E}$est *approprié* pour évaluer une expression simple e^f

  — La fonction $\mathbb{F}_e(\mathcal{E}, e^f)$ trouve le bon environnement d'évaluation de l'expression simple e^f à partir de l'environnement en éliminant les tranches internes [10] non utiles. La tête de cet environnement est une tranche avec un paquet qui possède potentiellement plusieurs tuples et une liste d'expressions groupantes.

  Une fois le bon environnement d'évaluation trouvé, on évalue la sous-expression simple e^f pour chacun des tuples du paquet de la première tranche, en scindant en quelque sorte le groupe, et en fabriquant à chaque fois une tranche avec un paquet à un seul tuple. Enfin, une fois les valeurs récupérées, on leur applique l'interprétation du symbole d'agrégat pour obtenir une valeur.

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

10. Par tranche interne nous voulons désigner la tranche la plus récente.

$$\frac{c \in \mathcal{V}}{\mathbb{B}_u(G,c)} \qquad \frac{e^f \notin \mathcal{V} \quad e^f \in G}{\mathbb{B}_u(G,e^f)} \qquad \frac{(\mathtt{fn}(\overline{e^f})) \notin G \quad \bigwedge_{e^f \in \overline{e^f}} \mathbb{B}_u(G,e^f)}{\mathbb{B}_u(G,\mathtt{fn}(\overline{e^f}))}$$

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

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

$$\frac{\mathbb{F}_e(\mathcal{E},e^f) = \mathtt{undefined} \quad \mathbb{S}_e(A,\mathcal{E},e^f)}{\mathbb{F}_e(((A,G,T)::\mathcal{E}),e^f) = (A,G,T)::\mathcal{E}}$$

Figure III.9. *Sémantique de la fonction qui trouve le bon environnement d'évaluation d'une expression avec agrégat*

Sémantique des formules

Les formules dénotent des booléens abstraits ($\mathcal{A}Bool$). On détaille ci-dessous la sémantique dénotationnelle pour chacune des formes syntaxiques présentées dans figure III.5 :

— La sémantique de la formule `true` correspond à la valeur vrai des booléens abstraits dénotée $\top^{\mathcal{B}}$:

$$[\![\mathtt{true}]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} \quad = \quad \top^{\mathcal{B}}$$

— Le symbole de négation `not` est interprété en faisant une descente récursive puis en appliquant la négation des booléens abstraits à l'interprétation de la sous-formule :

$$[\![\mathtt{not}\ \ f]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} \quad = \quad \neg^{\mathcal{B}}[\![f]\!]^{\mathrm{b}}_{<i,\mathcal{E}>}$$

— Les symbole de conjonction et de disjonction sont interprétés par descente récursive puis en appliquant la conjonction $\wedge^{\mathcal{B}}$ ou la disjonction $\vee^{\mathcal{B}}$ de la logique abstraite aux interprétations des sous-formules.

$$\begin{array}{lcl} [\![f_1\ \ \mathtt{and}\ \ f_2]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} & = & [\![f_1]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} \wedge^{\mathcal{B}} [\![f_2]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} \\ [\![f_1\ \ \mathtt{or}\ \ f_2]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} & = & [\![f_1]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} \vee^{\mathcal{B}} [\![f_2]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} \end{array}$$

— Une liste d'expressions $(\overline{e_i})$ portée par un symbole de prédicat $\mathfrak{pr}$ est évaluée en appliquant son interprétation $[\![\mathfrak{pr}]\!]_p$ à la liste des valeurs que les expressions dénotent :

$$[\![\mathfrak{pr}(\overline{e_i})]\!]^{\mathrm{b}}_{<i,\mathcal{E}>} \quad = \quad [\![\mathfrak{pr}]\!]_p(\overline{[\![e_i]\!]^{\mathrm{a}}_{\mathcal{E}}})$$

— Pour les quantificateurs universel et existentiel, il faut que l'évaluation soit vraie pour tout les tuples de q pour le premier et pour au moins un tuple pour le deuxième.

Pour le reste, le principe de l'évaluation des expressions portées par un prédicat reste le même que pour un prédicat :

$$[\![\mathfrak{pr}(\overline{e_i},\ \texttt{all}\ q)]\!]^{\mathfrak{b}}_{<i,\mathcal{E}>} = \top^{\mathcal{B}} \text{ ssi } [\![\mathfrak{pr}(\overline{e_i})]\!]^{\mathfrak{b}}_{<i,(\ell(t),-,[t])::\mathcal{E}>} = \top^{\mathcal{B}} \text{ pour tout } t \in [\![q]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>}$$
$$[\![\mathfrak{pr}(\overline{e_i},\ \texttt{any}\ q)]\!]^{\mathfrak{b}}_{<i,\mathcal{E}>} = \top^{\mathcal{B}} \text{ ssi } [\![\mathfrak{pr}(\overline{e_i})]\!]^{\mathfrak{b}}_{<i,(\ell(t),-,[t])::\mathcal{E}>} = \top^{\mathcal{B}} \text{ pour un } t \in [\![q]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>}$$

— La formule $(\overline{e_i\ \texttt{as}\ a_i}\ \texttt{in}\ q)$ est vraie lorsque le tuple que dénote $(\overline{e_i\ \texttt{as}\ a_i})$ appartient à l'évaluation de q :

$$[\![\overline{e_i\ \texttt{as}\ a_i}\ \texttt{in}\ q]\!]^{\mathfrak{b}}_{<i,\mathcal{E}>} = \top^{\mathcal{B}} \qquad \text{si } (\overline{a_i = [\![e_i]\!]^{\mathfrak{a}}_{\mathcal{E}}}) \in [\![q]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>}$$

— La formule $\texttt{exists}\ q$ est vraie lorsque q contient au moins un tuple :

$$[\![\texttt{exists}\ q]\!]^{\mathfrak{b}}_{<i,\mathcal{E}>} = \top^{\mathcal{B}} \qquad \text{ssi } [\![q]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} \text{ est non vide}$$

Sémantique des requêtes

Le requêtes dénotent des multiensembles selon la sémantique dénotationnelle ci-dessous :

— La sémantique d'un nom de relation $\mathfrak{tbl}$ est simplement le multiensemble de tuple auquel ce nom de relation est associé. Ce multiensemble est obtenu avec la fonction dénotée $[\![\ ]\!]_{db}$:

$$[\![\mathfrak{tbl}]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} = [\![\mathfrak{tbl}]\!]_{db} \qquad \text{si } \mathfrak{tbl} \in \mathcal{T}ableName$$

— Les opérateurs `union` , `intersect` et `except` ont une sémantique multiensembliste et sont donc définis sur la multiplicité (le nombre d'occurrences) et non seulement sur l'appartenance. Pour l'union, on prend la somme des multiplicités pour chacun des éléments de l'un des deux multiensembles. L'intersection s'obtient avec la multiplicité minimale des éléments appartenant aux deux ensembles. Enfin, la différence correspond au multiensemble construit en prenant $n(t)$ fois chaque tuple t dans le premier multiensemble avec n égal au nombre d'apparitions de t dans le premier multiensemble moins le nombre d'apparitions dans le second.

$$\begin{aligned}
[\![q_1\ \texttt{union}\ q_2]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} &= [\![q_1]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} \cup_m [\![q_2]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} \\
[\![q_1\ \texttt{intersect}\ q_2]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} &= [\![q_1]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} \cap_m [\![q_2]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} \\
[\![q_1\ \texttt{except}\ q_2]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} &= [\![q_1]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} \setminus_m [\![q_2]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>}
\end{aligned}$$

— La jointure naturelle a la même définition que dans l'algèbre relationnelle sauf qu'elle est multiensembliste et qu'elle duplique les noms d'attributs communs :

$$[\![q_1 \bowtie q_2]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} = \left\{ \left| \left(\overline{a_i = c_i}, \overline{b_j = d_j}\right) \right| \begin{array}{l} (\overline{a_i = c_i}) \in [\![q_1]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} \wedge \\ (\overline{b_j = d_j}) \in [\![q_2]\!]^{\mathfrak{q}}_{<i,\mathcal{E}>} \wedge \\ (\forall\ i,j,\ a_i = b_j \implies c_i = d_j) \end{array} \right| \right\}$$

— La projection est définie sur des expressions nommées ($\overline{e_i \texttt{ as } a_i}$) et non seulement sur des noms d'attributs. Elle est calculée en insérant un à un dans l'environnement les tuples de la sous-requête [11] pour évaluer les expressions pour chaque tuple et obtenir ainsi un nouveau tuple à partir des évaluations des expressions associées aux noms. Le résultat d'une projection a la même cardinalité que la sous-requête :

$$[\![\pi_{(\overline{e_i \texttt{ as } a_i})}(q)]\!]^{\text{q}}_{<i,\mathcal{E}>} \quad = \quad \{\!|(\overline{a_i = [\![e_i]\!]^{\text{a}}_{(\ell(t),-,[t])::\mathcal{E}}}) \mid t \in [\![q]\!]^{\text{q}}_{<i,\mathcal{E}>}|\!\}$$

— Le mécanisme qui permet d'évaluer la sélection est semblable à celui de la projection. On introduit les tuples de la sous-requête dans l'environnement un à un et on évalue la formule pour ce tuple et on *filtre* : si la formule est vraie on ajoute le tuple courant au résultat final. Dans le cas contraire on ignore le tuple :

$$[\![\sigma_f(q)]\!]^{\text{q}}_{<i,\mathcal{E}>} \quad = \quad \{\!|t \in [\![q]\!]^{\text{q}}_{<i,\mathcal{E}>} \mid [\![f]\!]^{\text{b}}_{<i,(\ell(t),-,[t])::\mathcal{E}>} = \top|\!\}$$

— L'opérateur γ prend en arguments une liste d'expressions nommées ($\overline{e_j \texttt{ as } a_j}$), une liste d'expressions groupantes ($\overline{e_i}$), une formule (f) et une sous-requête (q). On commence par construire la partition multiensemble que la sous-requête dénote selon les expressions groupantes. Ensuite, pour chaque partition, qu'on dénote $\mathbb{P}_k(\overline{e_i}, [\![q]\!]^{\text{q}}_{<i,\mathcal{E}>})$, on ne garde que les tuples pour lesquels la formule est vraie, puis on projette sur les expressions nommées. Si la requête est bien formée, on retrouve les expressions groupantes ($\overline{e_i}$) dans les expressions nommées et parmi les expressions nommées ($\overline{e_j \texttt{ as } a_j}$), au moins une, de ceux n'apparaissant pas dans ($\overline{e_i}$), est avec agrégat

$$[\![\gamma_{(\overline{e_j \texttt{ as } a_j}, \overline{e_i}, f)}(q)]\!]^{\text{q}}_{<i,\mathcal{E}>} = \left\{\!\!\left| \overline{\left(a_j = [\![e_j]\!]^{\text{a}}_{(\ell(T_2),\overline{e_i},T_2)::\mathcal{E}}\right)} \right| T_2 \in \left\{\!\!\left| T_1 \in \mathbb{P}_k(\overline{e_i}, [\![q]\!]^{\text{q}}_{<i,\mathcal{E}>}) \right| [\![f]\!]^{\text{b}}_{<i,(\ell(T_1),\overline{e_i},T_1)::\mathcal{E}>} = \top \right\}\!\!\right\}$$

III.1.2.6. Preuve de concept : Une instance de la spécification

Nous présentons ici l'instance de la spécification utilisée pour le compilateur de SQL_{Coq} vers SQL_{Alg}. C'est également cette instance que nous utilisons dans la traduction certifiée que nous proposons dans notre thèse. Elle n'est pas très riche mais elle est très raisonnable et permet d'avoir une algèbre tout à fait fonctionnelle.

— Les valeurs peuvent être les booléens `true` et `false`, des entiers, des flottants ou des chaînes de caractères. Le code qui suit correspond au type inductif qui définit les valeurs. le type option nous permet de représenter des valeurs nulles typées. Par exemple la valeur nulle des entiers sera représentée par `Value_Z None`, et toute valeur `x` entière est représentée par `Value_Z (Some x)`.

11. Voir III.1.2.4. Les environnements de SQL_{Alg} .

```
Inductive value : Set :=
| Value_string : option string → value
| Value_Z : option Z → value
| Value_bool : option bool → value
| Value_float : option float → value.
```

— Les symboles de fonction binaire définis sont l'addition, la soustraction et la multiplication, chacun en deux versions, l'une pour les entiers, l'autre pour les flottants. Un symbole de fonction unaire est défini et c'est l'opposé d'un nombre ($-x$). Ce symbole existe lui aussi en deux versions, l'une pour les entiers et l'autre pour les flottants. Dans ce qui suit, nous étudions le code Coq correspondant à l'instanciation des symboles de fonction. Le type inductif `symbol` permet de construire des symboles de fonction à partir de chaînes de caractères. La fonction `interp_symbol` prend en arguments un symbole et une liste de valeurs et leur applique la sémantique du symbole. On remarque, par exemple, que la multiplication des flottants est représentée par `Symbol _ "mult."`. En regardant de plus près l'interprétation du symbole `Symbol _ "plus"`, il est possible de constate que si la liste de valeurs en argument contient exactement deux éléments non nulles, leur somme (`Zplus`) renvoyée et dans les autres cas, la valeur entière nulle (`Value_Z None`) est renvoyée. La dernière ligne de la fonction renvoie une valeur par défaut si le symbole en argument n'existe pas.

```
Inductive symbol : Type := | Symbol : string → symbol.
Definition interp_symbol f :=
  match f with
  | Symbol _ "plus" ⇒
    fun l ⇒ match l with
            | Value_Z (Some a1) :: Value_Z (Some a2) :: nil ⇒
              Value_Z (Some (Zplus a1 a2))
            | _ ⇒ Value_Z None  end
  | Symbol _ "plus." ⇒ (* ... *)
  | Symbol _ "mult" ⇒ (* ... *)  | Symbol _ "mult." ⇒ (* ... *)
  | Symbol _ "minus" ⇒ (* ... *) | Symbol _ "minus." ⇒ (* ... *)
  | Symbol _ "opp" ⇒ (* ... *)   | Symbol _ "opp." ⇒ (* ... *)
  | _ ⇒ fun _ ⇒ Value_Z None (* Valeur par defaut *)
  end.
```

— Les symboles d'agrégat qui existent dans cette instance sont :

  — `count` : cet opérateur renvoie le nombre d'éléments d'une collection quelques soient leurs types.

  — `sum` : cet opérateur renvoie la somme d'une liste d'entiers. Une deuxième version renvoie la somme d'une liste de flottants.

  — `max` : cet opérateur renvoie la plus grande valeur d'une liste d'entiers. Une deuxième version renvoie la valeur maximale d'une liste de flottants.

  — `avg` : cet opérateur renvoie la moyenne des valeurs d'une liste d'entiers. Une

deuxième version renvoie la moyenne d'une liste de flottants.

La formalisation Coq est similaire aux symboles de fonction. On peut remarquer que l'opérateur `count` renvoie le nombre d'éléments de la liste sans se soucier du type. Par contre, l'opérateur `sum` s'assure qu'il s'applique à des entiers [12].

```
Inductive aggregate : Type := | Aggregate : string → aggregate.
Definition interp_aggregate a l :=
  match a with
  | Aggregate "count" ⇒ Value_Z(Some(Z_of_nat(List.length l)))
  | Aggregate "sum" ⇒
    Value_Z (Some (fold_left
                     (fun acc x ⇒ match x with
                                  | Value_Z (Some x) ⇒ (acc + x)%Z
                                  | _ ⇒ acc end) l 0%Z))
  | Aggregate "sum." ⇒  (* ... *)
  | Aggregate "max" ⇒ (* ... *)  | Aggregate "max." ⇒ (* ... *)
  | Aggregate "avg" ⇒ (* ... *)  | Aggregate "avg." ⇒ (* ... *)
  | Aggregate _ ⇒ Value_Z None (* Valeur par defaut *)
  end.
```

— Les symboles de prédicat disponibles sont

  — <, <=, > et >= : pour les entiers et pour les flottants.

  — = (et ≠) : pour tous les types [13].

```
Inductive predicate : Type := | Predicate : string → predicate.
Definition interp_predicate p :=
  match p with
  | Predicate "<" ⇒
    fun l ⇒
      match l with
      | Value_Z (Some a1) :: Value_Z (Some a2) :: nil ⇒
        match Z.compare a1 a2 with Lt ⇒ true3 | _ ⇒ false3 end
      | _ ⇒ unknown3 (* Valeur par defaut *)
      end
  | Predicate "<." ⇒ (* ... *)
  | Predicate "≤" ⇒ (* ... *)  | Predicate "≤." ⇒ (* ... *)
  | Predicate ">" ⇒ (* ... *)  | Predicate ">." ⇒ (* ... *)
  | Predicate "≥" ⇒ (* ... *)  | Predicate "≥." ⇒ (* ... *)
  | Predicate "=" ⇒ (* ... *)  | Predicate "≠" ⇒ (* ... *)
  end
| _ ⇒ fun _ ⇒ unknown3 (* Valeur par defaut *)
end.
```

12. L'opérateur est défini en utilisant la fonction `fold` connue en programmation fonctionnelle.
13. Ce prédicat est défini par l'égalité syntaxique de Coq ce qui permet qu'il soit polymorphe.

Là aussi, les symboles de prédicat sont construits à partir de chaînes de caractères. L'interprétation d'un prédicat, par exemple `Predicate "<"`, renvoie `true3` [14] ou `false3`, lorsque la liste d'arguments contient exactement deux entiers et qu'elle renvoie `unknown3` lorsque le nombre d'arguments est différent de 2.

III.2. Q*Cert

III.2.1. Présentation de Q*Cert

Q*Cert [Auerbach *et al.*, 2017d] est une librairie Coq permettant la spécification, la vérification et l'implémentation de compilateurs de requêtes. Elle fait le lien entre différents formalismes de données. On trouve parmi ces représentations, l'algèbre relationnelle imbriquée (NRA). On trouve également NRA^e [Auerbach *et al.*, 2017b] qui est une extension de NRA permettant une meilleure gestion des environnements en introduisant des opérateurs permettant de modifier l'environnement. Q*Cert offre également des optimisations certifiées de requêtes.

III.2.2. NRA^e

Auerbach et ses co-auteurs proposent en 2017 dans [Auerbach *et al.*, 2017b] une algèbre imbriquée étendue qui prend en charge la gestion des environnements au niveau même du langage. Cette algèbre est mécanisée en Coq. Une preuve d'équivalence de sémantique avec une formalisation en Coq de NRA basée sur les combinateurs est établie. Ce langage reprend tout les opérateurs de NRA et permet l'accès et la modification de l'environnement au niveau syntaxique. NRA^e a été créé avec l'optique d'offrir un langage pouvant accueillir de manière élégante les langages centrés données ayant une gestion sophistiquée et subtile des environnements. Ceci est précisément le cas de SQL_{Alg} (et de SQL) et justifie le choix de NRA^e pour accueillir SQL_{Alg}.

III.2.2.1. Représentation des données

Le modèle de données de NRA^e regroupe différents types de valeurs au sein d'un seul type global `data`. Les valeurs peuvent être des constantes (c) de type entier, flottant, booléen ou chaîne de caractère. On remarque ici que contrairement à SQL_{Alg} et à l'approche présentée dans II.3.3.1, les valeurs ne sont pas génériques. Les valeurs peuvent également être des enregistrements ($\{\overline{A_i : d_i}\}$) ou des collections [15] ($[\overline{d_i}]$). De plus, deux constructeur `left` et `right` permettent de construire une valeur de type *union étiquetée* [16]. Enfin, le type `data` comprend une valeur nulle dénotée par `unit`.

14. Dénotation de la valeur vraie de la logique trivaluée
15. Des multiensembles.
16. Similaire au `sum type` de Coq.

$$d \quad ::= \quad c \mid \{\} \mid \{\overline{A_i : d_i}\} \mid \emptyset \mid [\overline{d_i}] \mid \texttt{left}\ d \mid \texttt{right}\ d \mid \texttt{unit}$$

Figure III.10. *Modèle de données de NRA^e*

```
Inductive data : Set :=
(* Valeur c *)
| dbool : bool → data
| dstring : string → data
| dnat : Z → data
| dfloat : float → data
(* Collection ∅ | [d_i] *)
| dcoll : list data → data
(* Enregistrement {} | {A_i : d_i} *)
| drec : list (string * data) → data
(* Type union etiquetee left d | right d *)
| dleft : data → data
| dright : data → data
(* Valeur nulle unit *)
| dunit : data
(* Le modele est extensible a d'autres types de donnees *)
| dforeign : foreign_data_model → data.
```

Figure III.11. *Définition du modèle de données de NRA^e sous Coq*

Sous Coq, le modèle de données de NRA^e est défini par un type inductif comme on peut le voir sur la figure III.11. Il est important ici de remarquer que :

— Les collections sont définies par le constructeur `dcoll` et une liste de `data`.

— Les enregistrements sont construit par `drec` et une liste de paires. Les noms d'attributs sont représentés par des éléments de type `string`.

— Le modèle est extensible via le constructeur `dforeign` et la définition d'une spécification `foreign_data_model`.

III.2.2.2. Syntaxe de NRA^e

La figure III.12 présente le fragment syntaxique de NRA^e qui a été utilisé dans la traduction. Voici quelques commentaires sur les différentes formes de requêtes :

— Une requête peut être une constante d de type `data`.

— La requête `get` permet de récupérer la collection associée à un nom donnée depuis l'instance de la base de données.

$$
\begin{aligned}
query \quad ::= \quad & d \mid \texttt{get}\ name \mid \texttt{In} \mid \texttt{Env} \mid query \circ^{e} query \mid \boxplus query \mid query \boxtimes query \\
& \mid query \circ query \mid query \times query \mid \chi_{\langle query \rangle}(query) \mid \sigma_{\langle query \rangle}(query) \\
& \mid query \oplus query
\end{aligned}
$$

FIGURE III.12. *Syntaxe de NRA^e*

— Elle peut être le mot clé `In` qui correspond à la représentation syntaxique de la valeur courante d'évaluation [17].

— Le mot clé `Env` est la représentation syntaxique de l'environnement d'évaluation courant. L'opérateur $\circ^e$ permet de modifier cet environnement.

— Elle peut également être une requête portée par un symbole unaire dénoté par $\boxplus$. Cette dénotation regroupe tous les symboles unaires de NRA^e, les symboles d'agrégat comme le `count` ou le `sum`, mais aussi la négation ou l'opérateur (`dot` *label*) qui permet d'accéder à la valeur associée à un nom d'attribut dans un enregistrement.

— Une autre forme est l'opérateur binaire représenté par $\boxtimes$ appliqué à deux requêtes. Avec le même principe que le point précédent, ce symbole regroupe tous les opérateurs binaires. On peut citer les opérateurs arithmétiques habituels comme l'addition et la soustraction, les opérateurs logiques comme conjonction et la disjonction ou les opérateurs ensemblistes comme l'union ou l'intersection.

— On peut composer deux requêtes avec l'opérateur $\circ$. Cet opérateur à une sémantique similaire à la composition de fonctions [18].

— Il est est également possible de faire le produit cartésien de deux requêtes avec le symbole $\times$

— $\chi_{\langle\ \rangle}(\ )$ a une sémantique très similaire à celle de la fonction `map` bien connue dans la programmation fonctionnelle [19].

— $\sigma_{\langle\ \rangle}(\ )$ permet de filtrer les éléments d'une collection pour lesquels une formule est vraie.

— L'opérateur $\oplus$ (`either`) s'applique à deux requêtes en partant obligatoirement d'une valeur courante de type union étiquetée. Il évalue la première requête si la valeur courante est de la forme (`left _`) et la deuxième si la valeur courante est de la forme (`right _`).

17. Voir III.2.2.3. Sémantique de NRA^e .
18. Voir III.2.2.3. Sémantique de NRA^e .
19. Voir I.4.2. Les opérateurs de NRA.

III.2.2.3. Sémantique de NRA^e

Les requêtes (ou expressions) NRA^e dénotent des valeurs de type `data`. La sémantique opérationnelle de NRA^e est présentée en Annexe C. On dénote la fonction d'évaluation de NRA^e par $[[q]]^{\text{NRA}^e}_{<db,\gamma,d>}$ où q est une requête, γ est un environnement local d'évaluation et d est la valeur courante d'évaluation [20]. Cette valeur courante est utilisée par le mécanisme des combinateurs pour passer le résultat d'une requête à une autre requête ou pour parcourir tous les éléments d'une collection. Ci-dessous, nous présentons cette sémantique dénotationnelle accompagnée de commentaires que nous jugeons utiles à la compréhension de la contribution de notre thèse. Cette sémantique est plus proche de la mécanisation en Coq :

— L'évaluation d'une constante correspond à la valeur de type *data* qu'elle représente :

$$[[c]]^{\text{NRA}^e}_{<db,\gamma,d>} = c$$

— La requête `get` permet de récupérer la collection associée à un nom donnée depuis l'instance de la base de données.

$$[[\texttt{get "t"}]]^{\text{NRA}^e}_{<\{...,\texttt{"t"}:T,...\},\gamma,d>} = T$$

— `In` est la représentation syntaxique de la valeur courante :

$$[[\texttt{In}]]^{\text{NRA}^e}_{<db,\gamma,d>} = d$$

— L'évaluation de `Env` est la valeur de l'environnement courant.

$$[[\texttt{Env}]]^{\text{NRA}^e}_{<db,\gamma,d>} = \gamma$$

— L'opérateur $\circ^e$ permet d'évaluer une requête avec l'évaluation d'une autre requête comme environnement d'évaluation. Concrètement, dans la définition ci-dessous, q_1 est évaluée et la valeur obtenue remplace l'environnement actuel, puis q_2 est évalué sous ce nouvel environnement :

$$[[q_2 \circ^e q_1]]^{\text{NRA}^e}_{<db,\gamma,d>} = [[q_2]]^{\text{NRA}^e}_{<db,[[q_1]]^{\text{NRA}^e}_{<db,\gamma,d>},d>}$$

— Pour évaluer une requête portée par un symbole unaire dénoté, on fait une descente récursive puis on applique l'évaluation du symbole à l'évaluation de la sous-requête :

$$[[\boxplus\, q]]^{\text{NRA}^e}_{<db,\gamma,d>} = [[\boxplus]]^{\text{uop}}[[q]]^{\text{NRA}^e}_{<db,\gamma,d>}$$

20. On l'appelle aussi valeur de retour.

— Pour évaluer un opérateur binaire appliqué à deux sous-requêtes, on fait une descente récursive puis on applique l'évaluation du symbole aux évaluations des deux sous-requêtes :

$$[[q_1 \boxtimes q_2]]^{\text{NRA}^e}_{<db,\gamma,d>} = [[q_1]]^{\text{NRA}^e}_{<db,\gamma,d>}[[\boxtimes]]^{\text{bop}}[[q_2]]^{\text{NRA}^e}_{<db,\gamma,d>}$$

— L'évaluation de la requête de la forme $q_2 \circ q_1$ est l'évaluation de q_2 avec comme valeur courante l'évaluation de q_1 :

$$[[q_2 \circ q_1]]^{\text{NRA}^e}_{<db,\gamma,d>} = [[q_2]]^{\text{NRA}^e}_{<db,\gamma,[[q_1]]^{\text{NRA}^e}_{<db,\gamma,d>}>}$$

— Le produit cartésien de NRA^e a la même sémantique que dans NRA [21]. Dans la définition qui suit, le symbole $\odot$ dénote la concaténation de tuples :

$$\begin{array}{lcl}
[[\emptyset \times q_2]]^{\text{NRA}^e}_{<db,\gamma,d>} & = & \emptyset \\
[[q_1 \times \emptyset]]^{\text{NRA}^e}_{<db,\gamma,d>} & = & \emptyset \\
[[q_1 \times q_2]]^{\text{NRA}^e}_{<db,\gamma,d>} & = & [d_1 \odot d_2] \cup [[[d_1] \times s_2]]^{\text{NRA}^e}_{<db,\gamma,d>} \cup [[s_1 \times ([d_2] \cup s_2)]]^{\text{NRA}^e}_{<db,\gamma,d>} \\
 & & \text{si } [[q_1]]^{\text{NRA}^e}_{<db,\gamma,d>} = [d_1] \cup s_1 \text{ et } [[q_2]]^{\text{NRA}^e}_{<db,\gamma,d>} = [d_2] \cup s_2
\end{array}$$

— Pour que la requête $\chi_{\langle q_1 \rangle}(q_2)$ ait du sens, il faut que l'évaluation de q_2 soit une collection de *data*. Appelons cette collection l. Le résultat de l'évaluation de la requête globale correspond à la collection contenant les évaluations de q_1 avec chacun des éléments de l comme valeur courante. On peut comprendre que dans la requête q_1 le mot-clé `In` correspond à un élément de l'évaluation de q_2. Cela nous permet d'appliquer des requêtes itérativement aux éléments d'une collection. Par exemple : l'évaluation de $\chi_{\langle \texttt{In} \rangle}(q)$ correspond à celle de q ou encore l'évaluation de $\chi_{\langle \texttt{neg In} \rangle}([\texttt{true}, \texttt{false}])$ donne $[\texttt{false}, \texttt{true}]$.

$$\begin{array}{lcll}
[[\chi_{\langle q_2 \rangle}(q_1)]]^{\text{NRA}^e}_{<db,\gamma,d>} & = & \emptyset & \text{si } [[q_1]]^{\text{NRA}^e}_{<db,\gamma,d>} = \emptyset \\
[[\chi_{\langle q_2 \rangle}(q_1)]]^{\text{NRA}^e}_{<db,\gamma,d>} & = & [[[q_2]]^{\text{NRA}^e}_{<db,\gamma,d_1>}] \cup [[\chi_{\langle q_2 \rangle}(s_1)]]^{\text{NRA}^e}_{<db,\gamma,d>} & \text{si } [[q_1]]^{\text{NRA}^e}_{<db,\gamma,d>} = [d_1] \cup s_1
\end{array}$$

La projection est un cas particulier de la requête $\chi_{\langle \rangle}()$ qui utilise l'opérateur unaire $r[l]$ qui est l'opérateur de restriction.

$$[[\pi_{\langle l \rangle}(q)]]^{\text{NRA}^e}_{<db,\gamma,d>} = [[\chi_{\langle \texttt{In}[l] \rangle}(q)]]^{\text{NRA}^e}_{<db,\gamma,d>}$$

— La sémantique de $\sigma_{\langle q_1 \rangle}(q_2)$ est similaire dans le principe au $\chi_{\langle \rangle}()$. La seule différence est que cette fois-ci nous avons également une contrainte sur q_1 puisque l'évaluation de cette requête doit être un booléen. Le résultat global est la collection des valeurs

21. Voir I.4.2. Les opérateurs de NRA.

pour lesquelles q_1 vaut `true`. Comme exemple, on peut prendre $\sigma_{\langle \texttt{true} \rangle}(q)$ dont la sémantique est égale à celle de q ou $\sigma_{\langle \texttt{neg In} \rangle}([\texttt{true}, \texttt{false}, \texttt{true}])$ qui vaut [`false`].

$$
\begin{array}{lcll}
[\![\sigma_{\langle q_2 \rangle}(q_1)]\!]^{\text{NRA}^e}_{<db,\gamma,d>} & = & \emptyset & \text{si } [\![q_1]\!]^{\text{NRA}^e}_{<db,\gamma,d>} = \emptyset \\
[\![\sigma_{\langle q_2 \rangle}(q_1)]\!]^{\text{NRA}^e}_{<db,\gamma,d>} & = & [\![\sigma_{\langle q_2 \rangle}(s_1)]\!]^{\text{NRA}^e}_{<db,\gamma,d>} & \text{si } [\![q_1]\!]^{\text{NRA}^e}_{<db,\gamma,d>} = [d_1] \cup s_1 \\
 & & & \text{et } [\![q_2]\!]^{\text{NRA}^e}_{<db,\gamma,d_1>} = \texttt{false} \\
[\![\sigma_{\langle q_2 \rangle}(q_1)]\!]^{\text{NRA}^e}_{<db,\gamma,d>} & = & [d_1] \cup [\![\sigma_{\langle q_2 \rangle}(s_1)]\!]^{\text{NRA}^e}_{<db,\gamma,d>} & \text{si } [\![q_1]\!]^{\text{NRA}^e}_{<db,\gamma,d>} = [d_1] \cup s_1 \\
 & & & \text{et } [\![q_2]\!]^{\text{NRA}^e}_{<db,\gamma,d_1>} = \texttt{true}
\end{array}
$$

— L'évaluation de $q_1 \oplus q_2$ s'effectue selon la valeur courante. Si la valeur courante est égale à une valeur `left` x, alors le résultat final est l'évaluation de q_1 avec la valeur courante x. Sinon, si la valeur courante initiale est égale à `right` y, alors le résultat final est l'évaluation de q_2 avec comme valeur courante y.

$$
\begin{array}{lcl}
[\![q_1 \oplus q_2]\!]^{\text{NRA}^e}_{<db,\gamma,\texttt{left } d>} & = & [\![q_1]\!]^{\text{NRA}^e}_{<db,\gamma,d>} \\
[\![q_1 \oplus q_2]\!]^{\text{NRA}^e}_{<db,\gamma,\texttt{right } d>} & = & [\![q_2]\!]^{\text{NRA}^e}_{<db,\gamma,d>}
\end{array}
$$

— On peut également citer l'opérateur `group by`, que nous utilisons dans ce travail, et qui est syntaxiquement défini à partir des opérateurs de NRA^e. Cet opérateur permet de construire la partition selon les valeurs d'un ensemble d'attributs [22].

$$
\begin{array}{l}
[\![\texttt{group by } g\ sl\ q]\!]^{\text{NRA}^e}_{<db,\gamma,d>} = \\
\quad (\chi_{\langle \texttt{In} \odot \{"g":(\chi_{\langle \texttt{In}[sl]=\texttt{Env.}"\$key" \rangle}(\texttt{Env.}"\$pregroup")) \circ^e (\{"\$key":\texttt{In}\} \odot \texttt{Env})\}\rangle} \\
\qquad \left(\texttt{distinct}(\pi_{\langle sl \rangle}((\texttt{Env.}"\$pregroup")))\right)) \\
\quad \circ^e (\{"\$pregroup" : q\})
\end{array}
$$

La définition de l'opérateur s'appuie sur l'environnement local pour fabriquer les différents groupes et les collecter dans une relation à un attribut g qui contient les différents groupes.

III.3. Conclusion

Le projet SqlCert propose un langage algébrique (SQL_{Alg}) qui capture la sémantique de SQL, en étendant l'algèbre relationnelle avec un opérateur qui permet d'exprimer les clauses `group by` et `having` et en proposant des environnements sophistiqués permettant de corréler les requêtes. Les langages que comprend Q*Cert sont divers et assez génériques pour capturer la sémantique d'un certain nombre de langages centrés données du monde réel. Parmi ces langages, NRA^e est très proche du langage théorique NRA, mais il dispose d'un environnement plat sous forme d'un enregistrement. De plus, NRA^e permet d'accéder

22. Voir I.4.2. Les opérateurs de NRA.

et de modifier l'environnement d'évaluation au niveau syntaxique. La modification des environnements au niveau syntaxique peut être très utile lors de la traduction de langages avec une gestion sophistiquée des environnements, ce qui est le cas de SQL_{Alg}. Cette possibilité de modification permet de refléter la gestion d'environnements définie dans l'évaluation du langage source dans la traduction vers NRA^{e}.

Cette thèse propose de faire le lien entre ces deux projets en introduisant une traduction de SQL_{Alg} vers NRA^{e}. La traduction est correcte. Elle est formalisée en Coq et son théorème de préservation de sémantique prouvé correct. Cette contribution, que nous exposons dans la partie suivante, a joué un rôle prédominant dans la construction de DBCert, qui est la première plate-forme d'interopérabilité entre langages centrés donnés.

Deuxième partie : Contribution

Est-ce que le langage SQL peut être directement compilé vers un langage centré données non relationnel ?

Pour répondre à cette question, nous avons choisi de partir de la mécanisation de SQL_{Alg} et de formaliser, toujours en Coq, une compilation *certifiée* vers NRA^e. Le compilateur est certifié, c'est à dire accompagné des preuves d'un ensemble de théorèmes de préservation de sémantique. La traduction supporte les expressions simples et les expressions avec agrégats, les valeurs nulles, tous les types de formules et reflète adéquatement la gestion des environnements.

En s'appuyant sur ce compilateur nous affirmons que :

Oui, le langage SQL peut être compilé vers le langage NRA.

Nous l'avons formellement démontré sous Coq puisque SQL_{Alg} a une sémantique formellement équivalente à celle de SQL et NRA^e est formellement équivalent à NRA.

Outre l'apport d'une réponse définitive à cette question théorique, cette contribution est décisive dans la construction du compilateur DBCert. DBCert permet l'exécution certifiée de requêtes SQL dans un langage comme JavaScript, et ouvre la voie à l'exécution dans d'autres langages. DBCert a le potentiel de devenir un *hub* de formalisation de langages centrés données complètement opérationnel. Nous présentons plus en détail le projet DBCert dans la partie Discussion.

Chapitre

IV

De SQL_{Alg} vers NRA^e

Ce chapitre aborde le travail de notre thèse avec un angle de vue formel.

IV.1 Objectifs et présentation de la méthodologie **68**
IV.1.1 Objectifs de la traduction 68
IV.1.2 Présentation générale de la traduction 69
IV.2 Encodage du modèle de données **70**
IV.2.1 Encodage des valeurs 71
IV.2.2 Encodage des noms d'attributs 71
IV.2.3 Encodage de la logique trivaluée 71
IV.2.4 Encodage des tuples 72
IV.2.5 Encodage des collections 73
IV.2.6 Récapitulatif de l'encodage des données 74
IV.3 Encodage des instances de bases de données **74**
IV.4 Encodage et gestion des environnements **75**
IV.5 La traduction de SQL_{Alg} **79**
IV.5.1 Les expressions 79
IV.5.1.1 Expressions simples 80
IV.5.1.2 Expressions complexes 83
IV.5.1.3 Listes d'expressions nommées 85
IV.5.2 Les formules 85
IV.5.2.1 Symboles de prédicat 85
IV.5.2.2 Opérateurs NRA^e de la logique trivaluée 87
IV.5.3 Les requêtes 89
IV.5.4 Exemple de traduction 92
IV.6 Réalisation des spécifications **92**
IV.6.1 Les valeurs nulles 93
IV.6.2 Valeurs de la logique trivaluée 94
IV.6.3 Les symboles de fonction 95
IV.6.4 Les symboles d'agrégat 96
IV.6.5 Les symboles de prédicat et les opérateurs logiques 96
IV.7 Théorèmes de correction de la traduction **98**
IV.8 Conclusion **101**

IV.1. Objectifs et présentation de la méthodologie

IV.1.1. Objectifs de la traduction

L'objectif de ce travail de recherche est de lier les projets SqlCert et Q*Cert en formalisant sous Coq une traduction certifiée correcte, qui traduit n'importe quelle requête SQL$_{\text{Alg}}$ en une expression (algébrique) NRA^e. La correction ici veut dire que si la requête d'origine *a du sens* alors l'expression résultat a la *même sémantique*. Une requête a du sens si elle est bien formée dans le sens de la bonne formation définie dans [Benzaken et Contejean, 2019].

Nous dénotons cette traduction de requêtes par $\mathcal{T}^{\text{Q}}(\ )$. Les requêtes de SQL$_{\text{Alg}}$ représentent des multiensembles de tuples et les expressions de NRA^e expriment des multiensembles de `data`. Nous dénotons l'encodage qui transforme un multiensemble de tuples en multiensemble de `data` par $\mathcal{E}^{\text{bag}}(\ )$.

En utilisant de ces deux fonction, en faisant le lien entre les représentations des données ($\mathcal{E}^{\text{db}}(\ )$), en partant d'un environnement d'évaluation vide ([]), et en prenant n'importe quelle valeur courante de NRA^e (d), le théorème de préservation sémantique est de la forme :

$$\forall i\ d\ q, [[\mathcal{T}^{\text{Q}}(q)]]^{\text{NRA}^e}_{<\mathcal{E}^{\text{db}}(i),[],d>} =_{bags} \mathcal{E}^{\text{bag}}([[q]]^{\text{q}}_{<i,[]>})$$

Une requête SQL$_{\text{Alg}}$ peut contenir des noms de relations, des expressions algébriques (sous-requêtes), des listes d'expressions nommées et/ou des formules. Coté NRA^e, tous ces éléments sont confondus et représentent des expressions algébriques. Chacune des briques syntaxiques qui construisent les requête SQL$_{\text{Alg}}$ doit être traduite en une expression NRA^e.

Les requête SQL$_{\text{Alg}}$ et les expressions NRA^e s'évaluent sur une instance dans laquelle se trouvent les données. Des deux cotés, cette instance est constante car les algèbres ne font qu'interroger les données et ne les modifient pas. Chacune des deux algèbres dispose d'un opérateur qui permet de récupérer l'extension d'une relation à partir de son nom. Il faut encoder l'instance SQL$_{\text{Alg}}$ dans une instance NRA^e et faire correspondre les opérateurs.

Les requêtes, formules et expressions SQL$_{\text{Alg}}$ sont évaluées sous des environnements sophistiqués qui évoluent durant l'évaluation. Les expressions NRA^e, quant à elles, sont évalués sous des environnements locaux classiques (sous forme d'enregistrements). Les environnements sources doivent être encodées en environnements cibles et leur évolution reflétée dans les traductions.

Les expressions de SQL$_{\text{Alg}}$ dénotent des valeurs potentiellement nulles. Le langage NRA^e ne supporte pas les valeurs nulles tel que définies dans SQL (et donc dans SQL$_{\text{Alg}}$). Ces valeurs doivent être encodées dans le modèle de données de NRA^e et les opérateurs qui prennent en compte ces valeurs nulles doivent être définis.

Les formules de SQL$_{\text{Alg}}$ dénotent des valeur de la logique trivaluée. Cette logique doit être encodée dans le modèle de données de NRA^e et ses opérateurs spécifiques sont à

définir dans le langage cible.

Les requêtes de SQL_{Alg} dénotent des multiensembles de tuples. Les tuples et les multiensembles doivent être encodés dans le modèle de données cible.

Pour récapituler, afin d'atteindre notre objectif, nous devons :

— Définir une traduction des requêtes, des expressions algébriques, des formules et des expressions vers des expressions NRA^e.

— Encoder les valeurs nulles et la logique trivaluée dans le modèle de données de NRA^e.

— Encoder les environnement et refléter comment ils sont manipulés dans SQL_{Alg}.

— Prouver en Coq les théorèmes de préservation de sémantique des traductions et des encodages.

Parallèlement, nous nous sommes imposés un objectif lié à la formalisation en Coq du langage . Cette formalisation est paramétrée par une spécification d'un certain nombre de composants génériques et de propriétés. Cette généricité doit être conservée après la traduction.

IV.1.2. Présentation générale de la traduction

Le langage source SQL_{Alg} est une extension de l'algèbre relationnelle qui prend en compte les subtilités sémantiques de SQL. Le langage cible NRA^e est une extension de NRA. La similarité des concepts théoriques qui fondent ces deux langages facilite certains aspects de la traduction. En effet, même brouillée par des choix méthodologiques et d'implémentation différents, la présence d'un fragment commun entre les deux langages est évidente. En faisant abstraction de ces différences, les opérateurs algébriques de SQL_{Alg} ont leurs équivalents dans NRA^e. Cependant, la traduction doit prendre en compte toutes ces différences tout en répondant aux objectifs cités précédemment.

La difficulté principale réside dans le fait de devoir répercuter la subtile gestion des environnements de SQL_{Alg} [1] dans la traduction. Cette gestion qui se fait au niveau de l'évaluation de SQL_{Alg}, est reflétée dans la traduction et donc explicitée au niveau syntaxique.

La figure IV.1 illustre un autre aspect de ce travail. En effet, nous traduisons des briques syntaxiques différenciées vers des expressions NRA^e. Le même principe s'applique aux domaines d'évaluation : durant l'encodage du modèle de données source, nous passons de plusieurs domaines d'évaluation différenciés au type `data` qui est le domaine d'évaluation des expressions NRA^e.

Une autre difficulté provient de la manière dont les expressions et les formules sont évaluées en prenant en compte les valeurs nulles et la logique trivaluée. Ceci est finement reflété dans la traduction.

1. Voir III.1.2.4. Les environnements de SQL_{Alg} .

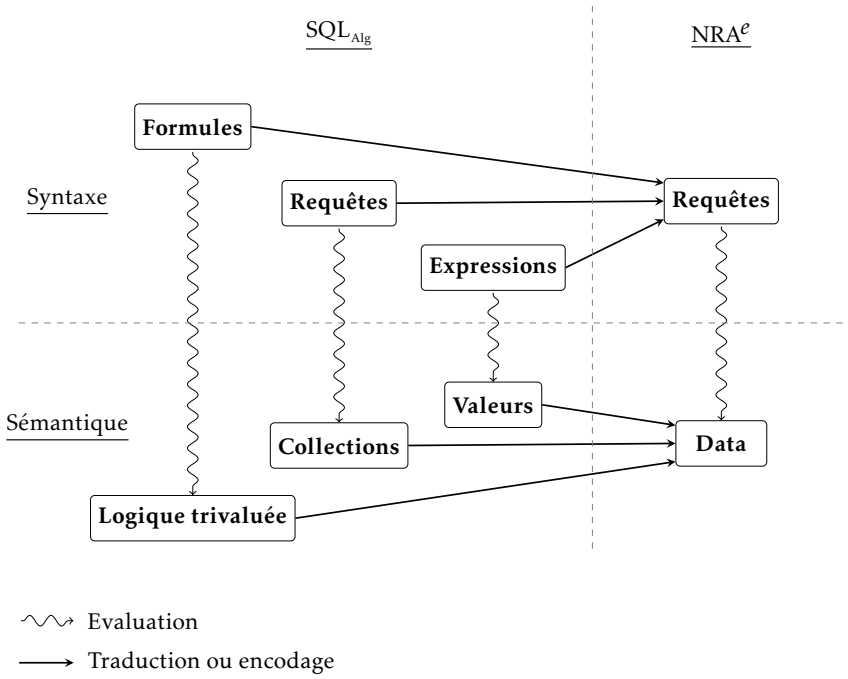


Figure IV.1. *Architecture de la traduction*

Enfin, un autre aspect a guidé nos choix dans la définition des traductions, il s'agit de conserver la généricité après la traduction. En effet, la formalisation de SQL$_{Alg}$ est paramétrée par une spécification qui définit plusieurs composants abstraits ainsi qu'un certain nombre de propriétés. Cette spécification est instanciée pour refléter tel ou tel système ou pour extraire un compilateur certifié. Le langage `ToyExp` [2] donne une intuition sur comment SQL$_{Alg}$ est formalisé. Dans le même esprit, et pour conserver cette généricité, la traduction est paramétrée par une spécification contenant la spécification source avec en plus des fonctions d'encodage ou des opérateurs abstraits NRA^e, mais aussi un certain nombre de propriétés nécessaires aux preuves de préservation de sémantique. Nous proposons également une instance de cette spécification en partant de la preuve de concept [3] définie dans [Benzaken et Contejean, 2019]. `ToyExpCert` [4] schématise l'approche adoptée pour construire notre compilateur.

IV.2. Encodage du modèle de données

Afin de définir un encodage qui permet de refléter tous les aspects du modèle de données plat de SQL$_{Alg}$, qui rappelons le, est paramétré par une spécification avec des

2. Voir II.3.3.1. Le langage `ToyExp`.
3. Voir III.1.2.6. Preuve de concept : Une instance de la spécification.
4. Voir II.3.3. Compilation certifiée en Coq d'un langage jouet.

composants génériques, nous définissons des fonctions faisant *correspondre* les composants génériques à des composants du modèle de NRA^e. La spécification contient également les propriétés nécessaires pour faire les démonstrations en Coq, des théorèmes de préservation de sémantique. Cette approche permet de conserver la généricité de SQL_{Alg}. Ainsi, la traduction est orthogonale aux différents systèmes utilisant SQL. Concrètement, les traductions des requêtes, des formules et des expressions sont paramétrées par un certain nombre de spécifications. En quelque sorte, ces spécifications mettent à disposition des boites vides qui représentent des composants et des propriétés supposées vraies sur ces composants. Ces composants doivent être définis dans l'étape de réalisation. Ces spécifications sont définies en utilisant le type `Class` de Coq.

IV.2.1. Encodage des valeurs

Nous dénotons l'encodage des valeurs génériques par $\mathcal{E}^{\text{val}}(\ )$. Cette fonction associe à chaque valeur source une valeur du modèle cible. Elle fait partie des spécifications paramétrant la traduction. Nous supposons que cette fonction d'encodage est une injection. Cette hypothèse fait partie de la spécification pour assurer que la fonction $\mathcal{E}^{\text{val}}(\ )$ définie dans l'*instance* est effectivement injective. Cette hypothèse est nécessaire pour prouver certaines propriétés utiles aux preuves de correction, comme l'injectivité de l'encodage des tuples par exemple. Dans la figure IV.2, qui présente une partie d'une des spécifications que toutes les traductions prennent en paramètres, l'encodage des valeurs est appelé `value_to_data` et la propriété d'injectivité est nommée `value_to_data_inj`. Le champ `TRcd` correspond à la spécification du modèle de donnée de SQL_{Alg}. Au niveau de la réalisation, nous utilisons une implémentation de `TRcd` et nous définissons les autres champs de la spécification `ToData`.

IV.2.2. Encodage des noms d'attributs

Pour l'encodage des noms d'attributs génériques, nous utilisons la dénotation $\mathcal{E}^{\text{att}}(\ )$. Cette fonction fait correspondre chaque nom d'attribut source à un nom d'attribut dans le modèle cible. Là aussi, pour les mêmes raisons que pour les valeurs, $\mathcal{E}^{\text{att}}(\ )$ fait partie de la spécification. L'hypothèse à poser pour cette fonction, pour prouver les propriétés utiles aux preuves de correction, est que $\mathcal{E}^{\text{att}}(\ )$ préserve l'ordre. Dans la figure IV.2, $\mathcal{E}^{\text{att}}(\ )$ correspond à la fonction `attribute_to_string` et la propriété de préservation d'ordre au lemme `attribute_to_string_compare_lt`.

IV.2.3. Encodage de la logique trivaluée

La logique trivaluée est le domaine d'évaluation des formules et ne fait vraiment partie du modèle de données de SQL_{Alg}, mais il faut encoder ses valeurs au même titre que les données. Nous employons la dénotation $\mathcal{E}^{\text{B}}(\ )$ pour la fonction encodant les trois valeurs de cette logique. Comme on peut le voir sur la figure IV.3, qui présente une spécification

```
Class ToData (* ... *) : Type :=
  mk_C {
      TRcd : Rcd;
      (* ... *)
      attribute_to_string : attribute TRcd → string;
      attribute_to_string_compare_lt : ∀a1 a2,
          a1 < a2  → attribute_to_string a1 < attribute_to_string a2;
      value_to_data : value TRcd  → data;
      value_to_data_inj : ∀a1 a2,
          value_to_data a1 = value_to_data a2 →  a1 = a2;
      (* ... *)
    }.
```

FIGURE IV.2. *Spécification en Coq de l'encodage du modèle de donnée*

dont dépend la traduction des formules et des requêtes, nous définissons une fonction `boolB_to_data` qui correspond à $\mathcal{E}^{B}(\ )$. Le paramètre (B : Bool.Rcd) est également un paramètre de la formalisation de SQL$_{Alg}$. Nous posons aussi une valeur `dtrue` de type `data` et nous ajoutons une propriété qui fait correspondre la valeur (Bool.true B) du modèle de SQL$_{Alg}$ à la valeur posée dtrue. Une des formules de SQL$_{Alg}$ est la représentation de la valeur logique `true`, c'est pour cette raison que nous avons besoin de fixer une valeur qui lui correspond dans le modèle cible, afin de l'utiliser dans la traduction. La propriété sert à prouver la correction de cette partie de la traduction.

```
Class BoolBToData  (* ... *)  (B : Bool.Rcd) :=
  mk_C
    {
      dtrue : data;
      boolB_to_data : Bool.b B → data;
      boolB_to_data_true : boolB_to_data (Bool.true B) = dtrue;
      (* ... *)
    }.
```

FIGURE IV.3. *Spécification en Coq de l'encodage de la logique trivaluée*

IV.2.4. Encodage des tuples

Les tuples dans SQL$_{Alg}$ sont abstraits. Un tuple est défini par un *support* [5] et par une fonction qui associe à chaque nom d'attribut appartenant au support une valeur. Un tuple a également une représentation canonique sous forme d'une liste de paires (nom d'attribut/valeur). Dans le modèle de NRA^e, les enregistrements sont représentés par

5. Le support d'un tuple est l'ensemble des noms de ses attributs.

une liste de paires (`string`/`data`). Ainsi, en utilisant l'encodage des valeurs ($\mathcal{E}^{\text{val}}(\ )$) et celui des noms d'attributs ($\mathcal{E}^{\text{att}}(\ )$), nous construisons un enregistrement de type `data`. Cet fonction est dénotée par $\mathcal{E}^{\text{t}}(\ )$. La figure IV.4 montre la définition Coq de la fonction $\mathcal{E}^{\text{t}}(\ )$: `tuple_as_data_rec`. La fonction `tuple_as_data_rec` correspond à la forme canonique du tuple `t`. Le lemme `tuple_as_data_rec_eq` est une propriété qui a permis de prouver beaucoup de lemmes intermédiaires. Pour prouver ce lemme, nous utilisons la propriété d'injectivité de l'encodage des valeurs et la propriété de préservation d'ordre de l'encodage des noms d'attributs.

```
Definition tuple_as_data_rec (t: tuple) :=
  map
    (fun x ⇒ (attribute_to_string (fst x), value_to_data (snd x)))
    (tuple_as_pairs t).
Lemma tuple_as_data_rec_eq :
  ∀(t1 t2: tuple),
    t1 =t= t2 ↔
    tuple_as_data_rec t1 = tuple_as_data_rec t2.
Proof.
  (* ... *)
Qed.
```

Figure IV.4. *Définition en Coq de l'encodage des tuples*

IV.2.5. Encodage des collections

Les multiensembles de tuples du modèle de SQL_{Alg} sont traduit en appliquant à chaque tuple du multiensemble la traduction $\mathcal{E}^{\text{t}}(\ )$. Cet encodage est représenté par $\mathcal{E}^{\text{bag}}(\ )$. Les multiensembles du modèle cible sont représentés par une liste d'éléments de type `data`, mais au niveau sémantique, cette liste est traitée comme un multiensemble. Sur la figure IV.5, la fonction `bagT_to_listD` correspond à $\mathcal{E}^{\text{bag}}(\ )$. La fonction `elements` renvoie tous les éléments d'un multiensemble dans une liste. Pour finir, la fonction `listT_to_listD` applique l'encodage des tuples à tous les tuple d'une liste. L'une des propriétés les plus utiles dans le processus des preuves est celle définie par le lemme `listT_to_listD` qui dit que le nombre d'occurrences de tout tuple dans une liste correspond à la multiplicité de l'encodage de ce tuple dans l'encodage de la liste. Nous utilisons cette définition car du coté NRA^{e}, l'équivalence entre deux collections de type `data` correspond à l'égalité de la multiplicité de tout enregistrement dans les deux listes qui implémentent ces deux collections. Du coté SQL_{Alg}, un lemme existant montre que deux *bags* sont équivalents si et seulement si le nombre d'occurrence de tout tuple est le même dans les listes d'éléments de chacun des *bags*.

```
Definition listT_to_listD (lt: list tuple) :=
  map (fun x ⇒  (drec (tuple_as_data_rec x))) lt.
Definition bagT_to_listD (b: bag) :=
  listT_to_listD (elements b).
Lemma nb_occ_mult_listT_eq :
  ∀(l: list tuple) (t: tuple),
    nb_occ t l = mult (listT_to_listD l) (drec (tuple_as_data_rec t)).
(* drec est le constructeur des enregistrements de type data. *)
Proof.
  (* ... *)
Qed.
```

FIGURE IV.5. *Définition en Coq de l'encodage des multiensembles de tuples*

IV.2.6. Récapitulatif de l'encodage des données

La TABLE IV.1 résume les fonctions d'encodage des différents types de données et de la logique trivaluée vers des données du modèle de NRA^e.

Type de données	Dénotation	Fonction Coq
Valeurs génériques	$\mathcal{E}^{\text{val}}(\,)$	`value_to_data`
Noms d'attributs	$\mathcal{E}^{\text{att}}(\,)$	`attribute_to_string`
Tuples abstraits	$\mathcal{E}^{\text{t}}(\,)$	`tuple_as_data_rec`
Multiensembles de tuples	$\mathcal{E}^{\text{bag}}(\,)$	`bagT_to_listD`
Logique trivaluée	$\mathcal{E}^{\text{B}}(\,)$	`boolB_to_data`

TABLE IV.1. *Encodages des données de SQL$_{Alg}$ vers le modèle de données de NRA^e*

IV.3. Encodage des instances de bases de données

La base de données est une instance du contenu des relations utilisée au moment de l'évaluation. Cette instance est donc nécessaire uniquement au moment de l'exécution et n'est pas utile pour la traduction. Cependant, pour les preuves de préservation de sémantique, nous devons savoir comment l'instance est représentée du coté SQL$_{\text{Alg}}$ et comment elle est encodée en instance équivalente du coté NRA^e. De plus, puisque nous nous intéressons uniquement aux requêtes de type `select`, l'instance de la base est constante tout au long de l'évaluation. Une instance (de base de données) SQL$_{\text{Alg}}$ est définie par un enregistrement étiqueté par des noms de relations et où les valeurs sont des multiensembles de tuples. Une instance NRA^e est simplement un enregistrement où les valeurs sont de type `data`. Il est donc naturel de faire le changement de représentation, simplement en traduisant les noms de relations en *labels* et les collections de tuples via

la fonction $\mathcal{E}^{\text{bag}}(\ )$. Cette transformation est dénotée par $\mathcal{E}^{\text{tab}}(\ )$. La figure IV.6 présente la spécification qui concerne l'instance de la base de données. Le paramètre `TD` est une spécification de l'encodage du modèle, nous lui faisons appel ici car l'encodage des multiensembles en dépend et que nous l'utilisons pour encoder les relations qui sont dans l'instance. Le paramètre `relname` permet de définir des noms de relations génériques. La spécification comprend une fonction d'encodage des noms de relations et la propriété d'injectivité de cette fonction. La fonction `instance_to_bindings` est la fonction dénotée par $\mathcal{E}^{\text{db}}(\ )$ et qui transforme une instance de données du contexte de SQL_{Alg} vers une instance de données du contexte de NRA^{e}.

```
Class InstanceToNRAEnv
      `{TD : TupleToData}
      {relname : Type}
(*...*) : Type :=
  mk_C {
      relname_to_string : relname → String.string;
      relname_to_string_inj :
        ∀x y : relname,
          relname_to_string x = relname_to_string y → x = y;
      (*...*)
    }.
Definition instance_to_bindings i :=
  map
    (fun x ⇒ match x with
              | (r, b) ⇒ (relname_to_string r, dcoll (bagT_to_listD b))
              end) i.
(* dcoll est le constructeur des multiensembles de type data. *)
```

Figure IV.6. *Spécification et définition en Coq de l'encodage de l'instance de la base de données*

IV.4. Encodage et gestion des environnements

Afin d'atteindre l'objectif de traduire les requêtes, nous devons définir des traductions des expressions algébriques, des formules et des expressions (agrégats et fonctions). Concernant les environnements, une requête s'évalue sous un environnement vide. Des tranches peuvent être ajoutées à cet environnement [6] tout au long de l'évaluation.

Pour refléter la manipulation des environnements dans la traduction, nous devons :

1. encoder les environnements SQL_{Alg} dans le contexte d'évaluation de NRA^{e} ;
2. considérer, au niveau syntaxique, que l'environnement local est un encodage correct

6. Voir III.1.2.4. Les environnements de SQL_{Alg} .

de l'environnement du coté source, et refléter syntaxiquement les différentes manipulations qui peuvent s'appliquer à l'environnement durant l'évaluation source dans la traduction.

Une tranche d'environnement de SQL_{Alg} possède, en plus du paquet concret de tuples ($T(S)$), une partie abstraite : les noms d'attributs de la tranche ($A(S)$) et les expressions groupantes ($G(S)$). Durant l'évaluation, la partie abstraite permet uniquement de manipuler l'environnement pour *se positionner* sur la *bonne* tranche pour évaluer tel ou tel élement. Une fois la *bonne* tranche trouvée, la partie concrète est utilisée pour évaluer un résultat. La manipulation de l'environnement doit être reflétée dans la traduction et la partie concrète accessible au niveau syntaxique.

L'environnement local de NRA^e est un enregistrement de type `data`. Constatons que :

— Les expressions groupantes sont des éléments syntaxiques non représentables en données `data`.

— La partie abstraite n'est plus utile une fois la manipulation de l'environnement reflétée dans la traduction. Autrement dit, la partie abstraite est statique et n'est plus nécessaire pendant l'évaluation.

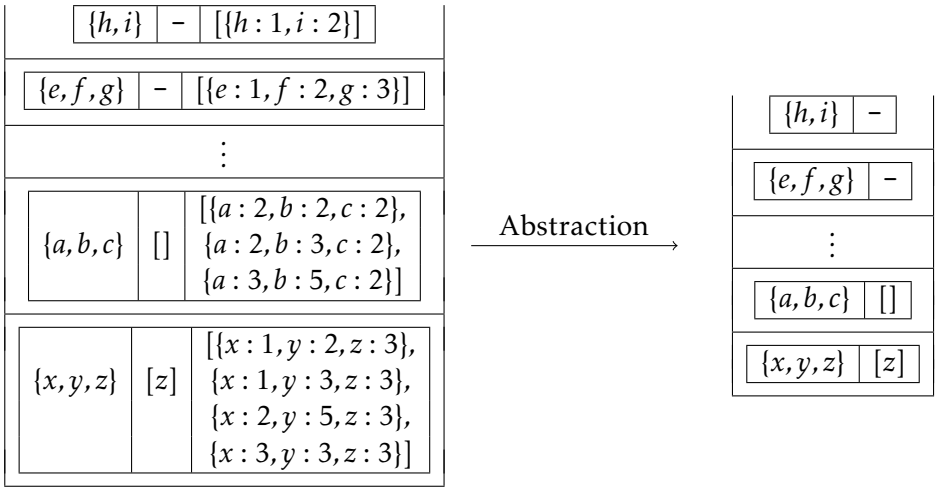


FIGURE IV.7. *Extraction de la partie abstraite d'un environnement*

Pour ces raisons, nous détachons la partie abstraite d'une environnement de sa partie concrète et nous encodons la partie concrète pour qu'elle soit utilisée durant l'évaluation. La partie abstraite, quant à elle, paramètre la fonction de traduction, mais n'est pas encodée. Plus concrètement, nous introduisons la notion de *tranche abstraite* qui correspond à la paire $S^a = (A, G)$ [7]. Nous introduisons également la notion d'*environnement abstrait* qui correspond à une pile de tranches abstraites. La FIGURE IV.7 illustre un exemple d'un

7. Voir III.1.2.4. Les environnements de SQL_{Alg} .

environnement SQL_{Alg} et de l'environnement abstrait correspondant.

Ainsi, nous découpons l'environnement en deux parties, l'une abstraite avec les informations nécessaires à la traduction (au niveau syntaxique) et l'autre concrète avec les paquets de tuples. En effet, au moment de la traduction, nous n'avons pas accès aux données concrètes, mais nous devons conserver la partie abstraite pour l'utiliser. On peut voir cela ainsi : l'environnement abstrait est *fixé* à la traduction et à chaque fois que nous modifions l'environnement coté source pour la prochaine étape d'évaluation, nous modifions l'environnement abstrait de la traduction de manière correspondante.

Concrètement, toutes nos traductions sont paramétrées par un environnement abstrait pour conserver les informations sur les noms d'attributs et les expressions groupantes des différentes tranches pour les utiliser dans la traduction.

Concernant la partie concrète des environnements, celle-ci est représentée dans un environnement local de $\text{NRA}^{\mathcal{e}}$ sous forme de pile. L'encodage d'un environnement SQL_{Alg} initial est définie par :

$$\begin{array}{lcl} \mathcal{E}^{\text{env}}([]) & = & \{\} \\ \mathcal{E}^{\text{env}}((A,G,T)::\texttt{env}) & = & \{"slice" : \mathcal{E}^{\text{bag}}(T), "tail" : \mathcal{E}^{\text{env}}(\texttt{env})\} \end{array}$$

```
Fixpoint env_to_data (e : env): data :=
  drec (* drec est le constructeur des enregistrements de type data *)
    (match e with
     | nil ⇒ nil
     | (_,lt)::tl ⇒
       ("slc",dcoll (listT_to_listD  lt))::("tl",env_to_data tl)::nil
     end).
```

Figure IV.8. *Définition en Coq de l'encodage de la partie concrète d'un environnement SQL_{Alg}*

Un environnement local est soit un enregistrement vide, soit un enregistrement à deux champs : le champ *"slice"* contient la collection de données (de type `data`) correspondant au sommet de la pile, tandis que le champ *"tail"* contient le reste de la pile. La figure IV.8 montre la définition de la fonction récursive `env_to_data` qui encode la partie concrète d'un environnement.

Cet encodage des partie concrètes des environnements n'est pas utile à la traduction. Cependant, définir cet encodage est impératif pour les preuves car plusieurs théorèmes sont énoncés pour un environnement SQL_{Alg} quelconque et l'encodage de cet environnement est explicité [8].

Une fois que nous avons défini l'encodage, nous devons également refléter la manipulation de l'environnement (ajout/suppression de tranches) durant l'évaluation. Cette

8. Voir IV.7. Théorèmes de correction de la traduction.

réflexion se fait au niveau syntaxique en supposant que l'environnement initial est bien encodé et que l'environnement abstrait correspond bien à l'environnement global.

Pour refléter les modifications de l'environnement nous définissons l'opération push, qui correspond à la création d'un nouvel enregistrement où *"slice"* contient la traduction de la collection qu'on veut ajouter et où *"tail"* contient l'ancien enregistrement. ainsi que l'opération pop, où le nouvel environnement est simplement le contenu *"tail"* du courant.

```
Definition push_in_nenv_then_eval slice exp : nraenv :=
  (AppEnv
     exp
     (Binop
        OpRecConcat
        (Unop  (OpRec "tl") Env)
        (Unop  (OpRec "slc") slice))).
```

Figure IV.9. *Définition de la fonction push qui ajoute une tranche à l'environnement courant*

La figure IV.9 présente la fonction (`push_in_nenv_then_eval slice exp`) qui ajoute la tranche (`slice`) à l'environnement courant (`Env` [9]) puis évalue l'expression (`exp`) sous ce nouvel environnement.

Cette fonction nous permet de définir dans le code les deux opérations suivantes qui modifient l'environnement de SQL$_{\text{Alg}}$ durant l'évaluation. Soit nous ajoutons ensemble tous les tuples correspondant à l'évaluation d'une sous-requête, soit nous les ajoutons séquentiellement, un à un, sous forme de singleton :

$$\begin{aligned} \texttt{push}_{bag}(exp) &= exp \circ^e \{(\text{"}slice\text{"} : \texttt{In}), (\text{"}tail\text{"} : \texttt{Env})\} \\ \texttt{push}_{one}(exp) &= exp \circ^e \{(\text{"}slice\text{"} : [\texttt{In}]), (\text{"}tail\text{"} : \texttt{Env})\} \end{aligned}$$

Dans les deux cas, cela revient à ajouter `In`. L'utilisation de `In` est liée à l'utilisation de l'opérateur $\chi_{\langle\,\rangle}(\,)$ de NRA^e [10]. Les deux fonctions push$_{bag}$ et push$_{one}$ permettent d'évaluer une expression après modification [11] de l'environnement de manière adéquate.

En résumé, les traductions dépendent de l'environnement abstrait qu'on peut obtenir syntaxiquement sur les requêtes et qui ne dépend pas de l'instance. L'environnement abstrait constitue ainsi un paramètre des traductions. La partie concrète est représentable et manipulable, mais elle n'est utile que pour l'évaluation. Pendant la traduction, les modifications de l'environnement doivent être reflétées pour les deux parties (concrète et abstraite).

9. Rappelons que l'environnement local courant est représenté syntaxiquement dans NRA^e par `Env`.
10. Voir III.2.2.3. Sémantique de NRA^e .
11. L'opérateur NRA^e $\circ^e$ permet la modification l'environnement avant l'évaluation d'une expression.

IV.5. La traduction de SQL_{Alg}

Les requêtes SQL_{Alg} s'écrivent avec différents composants syntaxiques : les expressions algébriques, les formules et les expressions. Chacun de ces composants est traduit en expression NRA^{e} :

— Les requêtes : $\mathcal{T}^{Q}(\ )$.

— Les expressions algébriques : $\mathcal{T}^{Q}_{\mathcal{A}}(\ )$.

— Les formules : $\mathcal{T}^{f}_{\mathcal{A}}(\ )$.

— Les expressions simples : $\mathcal{T}^{e^f}_{\mathcal{A}}(\ )$.

— Les expressions avec agrégats : $\mathcal{T}^{e^a}_{\mathcal{A}}(\ )$.

Hormis la traduction des requêtes, les autres traductions dépendent d'un environnement abstrait $\mathcal{A}$. La traduction des requêtes est un cas particulier de la traduction des expressions algébriques. En réalité, il s'agit de la traduction d'une expression algébrique sous un environnement abstrait vide : $\mathcal{T}^{Q}_{[]}(\ )$. Afin d'exprimer le fait qu'il s'agit d'une requête *top level*, et faire la différence avec une expressions algébrique quelconque qui aurait un environnement vide, nous omettons l'environnement abstrait dans les notations.

Nous présentons dans cette section les traductions de chacune des catégories syntaxiques, expressions simples et complexes, formules et expressions algébriques

IV.5.1. Les expressions

Dans SQL_{Alg}, on discrimine les expressions contenant des symboles d'agrégat des expressions sans aucun symbole d'agrégat . En effet, une expression sans agrégat, que nous appelons expression simple (dénotée e^f), s'évalue différemment d'une expression avec des symboles d'agrégat, que nous désignons par expression complexe et que nous dénotons par e^a. La différence se situe notamment au niveau de la gestion des environnements qui est subtile pour les expressions complexes.

Nous définissons une traduction des expressions simples ($\mathcal{T}^{e^f}_{\mathcal{A}}(\ )$) en expressions NRA^{e} et une traduction des expressions complexes ($\mathcal{T}^{e^a}_{\mathcal{A}}(\ )$) également en expressions NRA^{e}.

La formalisation des expressions simples est paramétrée par une spécification définissant des symboles de fonction génériques et celle des expressions complexes l'est par une spécification qui définit des symboles d'agrégat génériques. Nous appliquons la même approche que pour le modèle de données en définissant des spécifications de traductions *correctes* des symboles de fonction et des symboles d'agrégat.

Nous utilisons ces spécifications dans la définition de la traduction et nous nous servons de leurs propriétés de correction dans le processus de preuve. Pour obtenir une traduction opérationnelle [12], nous définissons une réalisation de chacune des spécifications

12. *i.e.*, qui est exécutable et qui calcule des résultats

en fournissant les définitions et les théorèmes de correction.

IV.5.1.1. Expressions simples

Les expressions simples dénotent des valeurs dans le modèle de SQL_{Alg}. Une expression simple est soit une valeur, soit un nom d'attribut, soit une liste d'expressions simples portée par un symbole de fonction.

```
Class SymbolToNRAEnv  `{TD : @ToData} : Type :=
  mk_C
    {
      (* ... *)
      symbol_to_nraenv : symbol →  list nraenv → nraenv;
      symbol_to_nraenv_ignores_did :
        ∀(s: symbol) (lnra: list nraenv),
          (∀(n: nraenv) i reg id1 id2,
              In n lnra →
              nraenv_eval i n reg id1 = nraenv_eval i n reg id2) →
          ∀i reg id1 id2,
            nraenv_eval i (symbol_to_nraenv s lnra) reg id1 =
            nraenv_eval i (symbol_to_nraenv s lnra) reg id2;
      symbol_to_nraenv_eq :
        ∀X (nraenv_of : X → nraenv) (val_of : X → value),
        ∀ i dreg, ∀(l:list X)
                     (∀(x: X),
                         In x l → ∀did,
                           nraenv_eval i (nraenv_of x) dreg did =
                           Some (value_to_data  (val_of x))) →
            ∀s did,
              nraenv_eval
                i (symbol_to_nraenv s (map nraenv_of l)) dreg did =
              Some (value_to_data (interp_symbol s (map val_of l)));
                     (* ... *)
    }.
```

Figure IV.10. *Spécification en Coq de la traduction des symboles de fonction*

La figure IV.10 montre la spécification de la traduction des symboles de fonction (dénotée par $\mathcal{T}^{\text{fn}}(\ ,\ )$). La fonction de traduction, appelée `symbol_to_nraenv`, prend en arguments un symbole de fonction et une liste d'expressions NRA^e et construit une expression NRA^e. La liste d'expressions correspond à une liste de traductions. La propriété `symbol_to_nraenv_ignores_did` dit que la traduction des symboles de fonction ne dépend pas de la valeur courante [13]. Une propriété de préservation de sémantique

13. Voir III.2.2.3. Sémantique de NRA^e .

est également admise dans la spécification. Nous reviendrons sur cette propriété dans la section IV.7.

La traduction des expressions simples est définie de la manière suivante :

— Pour traduire une valeur SQL_{Alg}, il suffit de l'encoder dans le modèle de $NRA^{\mathcal{E}}$:

$$\mathcal{T}^{ef}_{\mathcal{A}}(v) \quad = \quad \mathcal{E}^{\text{val}}(v)$$

— Un nom d'attribut est évalué à une valeur par défaut si l'environnement abstrait est vide. Cette situation ne se produit jamais car l'environnement n'est jamais vide si l'on part d'une requête. Elle est juste là pour nous permettre d'avoir une fonction totale.

$$\mathcal{T}^{ef}_{[]}(a) \quad = \quad \mathcal{E}^{\text{val}}(\texttt{default}) \qquad \text{Si } a \text{ est un nom d'attribut}$$

— Sous un environnement abstrait non vide, la traduction d'un nom d'attribut faisant partie des noms d'attribut de la tranche courante se fait de la manière suivante :

1. On trouve la valeur associée à *"head"* dans l'environnement local de $NRA^{\mathcal{E}}$ (équivalent à la partie concrète de l'environnement d'évaluation) ;
2. On prend le premier élément de ce que l'on obtient dans (a) ;
3. On encode l'attribut dans le modèle de $NRA^{\mathcal{E}}$;
4. On trouve dans la requête obtenue (b) la valeur associée à l'attribut encodé dans (c).

$$\mathcal{T}^{ef}_{(A,G)::\mathcal{A}}(a) \quad = \quad \texttt{first\_of}(\texttt{Env} \cdot \textit{"head"}) \cdot \mathcal{E}^{\text{att}}(a) \qquad \text{Si } a \in A$$
$$\texttt{first\_of} \text{ est un opérateur unaire de } NRA^{\mathcal{E}}$$

— Sous un environnement abstrait non vide, la traduction d'un nom d'attribut ne faisant pas partie des noms d'attribut de la tranche courante se fait de la manière suivante : On essaye de traduire l'attribut après avoir enlevé la tranche courante de l'environnement abstrait et avoir remplacé l'environnement local de $NRA^{\mathcal{E}}$ par uniquement la partie se trouvant dans son attribut *"tail"*.

$$\mathcal{T}^{ef}_{(A,G)::\mathcal{A}}(a) \quad = \quad \mathcal{T}^{ef}_{\mathcal{A}}(a) \circ^{e} (\texttt{Env} \cdot \textit{"tail"}) \text{ Si } a \notin A$$

— Une expression simple composée d'un symbole de fonction et d'une liste d'expressions simples est traduite en utilisant la traduction abstraite des symboles de fonction et la liste des traductions sous le même environnement abstrait des sous-expressions simples.

$$\mathcal{T}^{ef}_{\mathcal{A}}(\mathfrak{fn}\ (\overline{ef})) \quad = \quad \mathcal{T}^{\mathfrak{fn}}(\mathfrak{fn}, (\overline{\mathcal{T}^{ef}_{\mathcal{A}}(ef)}))$$

La figure IV.11 détaille la définition Coq de la traduction des expressions simples. La classe `FTermToNRAEnv` regroupe les spécifications nécessaires à la définition de la traduction des expressions simples.

```
Class FTermToNRAEnv
      `{TD : ToData}
      `{SN : SymbolToNRAEnv TD} : Type.

Fixpoint env_dot_to_nraenv (ne:nenv) a : nraenv :=
  match ne with
  | nil ⇒
    NRAEnvConst (value_to_data (default_value (type_of_attribute a)))
  | (s1,g1)::tl ⇒
    let hd :=
        NRAEnvHdDef (drec (tuple_as_dpairs (default_tuple s1)))
                    (NRAEnvUnop (OpDot "slc") NRAEnvEnv) in
    if a inS? s1 then (NRAEnvUnop (OpDot (attribute_to_string a)) hd)
    else
      NRAEnvAppEnv
        (env_dot_to_nraenv tl a) (NRAEnvUnop (OpDot "tl") NRAEnvEnv)
  end.
Fixpoint funterm_to_nraenv (ne:nenv) (f : funterm) : nraenv :=
  match f with
  | F_Constant c ⇒ NRAEnvConst (value_to_data c)
  | F_Dot  a ⇒ env_dot_to_nraenv ne a
  | F_Expr f l ⇒ symbol_to_nraenv f (map (funterm_to_nraenv ne) l)
  end.
```

FIGURE IV.11. *Spécification en Coq de la traduction des expressions simples*

IV.5.1.2. Expressions complexes

Les expressions complexes représentent des valeurs dans le modèle de SQL_{Alg}. Un expression complexe est soit une liste d'expressions complexes portée par un symbole de fonction, soit une expression simple portée par un symbole d'agrégat.

La figure IV.12 met en évidence la spécification de la traduction des symboles d'agrégat (représentée par $\mathcal{T}^{ag}(\ ,\ )$). La fonction de traduction, nommée `aggregate_to_nraenv`, prend en arguments un symbole d'agrégat et une expression NRA^e et construit une expression NRA^e. L'expression en argument correspond à une traduction. La propriété `aggregate_to_nraenv_ignores_did` dit que la traduction des symboles d'agrégat ne dépend pas de la valeur courante [14]. La spécification admet une propriété de correction et la propriété `interp_aggregate_permut` qui dit que l'interprétation d'un agrégat est la même pour deux listes d'arguments équivalentes modulo permutations.

```
Class AggregateToNRAEnv `{TD : ToData} : Type :=
  mk_C
    {
      aggregate_to_nraenv : aggregate → nraenv → nraenv;
      aggregate_to_nraenv_ignores_did :
        ∀s n i reg,
          (∀id1 id2,
              nraenv_eval i n reg id1 = nraenv_eval i n reg id2) →
          ∀id1 id2,
            nraenv_eval i (aggregate_to_nraenv s n) reg id1 =
            nraenv_eval i (aggregate_to_nraenv s n) reg id2;
      aggregate_to_nraenv_eq :
        ∀ag (a: funterm),
        ∀ft_to_nraenv ft_to_lval  br i dreg,
          (∀did, nraenv_eval br i (ft_to_nraenv a) dreg did =
                  Some
                    (dcoll (map value_to_data (ft_to_lval a)))) →
          ∀did,
            nraenv_eval
               i (aggregate_to_nraenv ag (ft_to_nraenv a))
              dreg did =
            Some (value_to_data (interp_aggregate ag (ft_to_lval a)));
      interp_aggregate_permut : ∀ag l1 l2,
          Permutation l1 l2 →
          interp_aggregate ag l1 = interp_aggregate ag l2;
    }.
```

Figure IV.12. *Spécification en Coq de la traduction des symboles d'agrégat*

14. Voir III.2.2.3. Sémantique de NRA^e .

La traduction des expressions complexes est définie comme suit :

— Une expression composée d'un symbole de fonction et d'une liste d'expressions complexes est traduite en utilisant la traduction abstraite des symboles de fonction et la liste des traductions sous le même environnement abstrait des sous-expressions.

$$\mathcal{T}^{e^a}_{\mathcal{A}}(\mathtt{fn}\ (\overline{e^a})) \quad = \quad \mathcal{T}^{\mathtt{fn}}(\mathtt{fn}, (\overline{\mathcal{T}^{e^a}_{\mathcal{A}}(e^a)}))$$

— Une expression commençant par un symbole d'agrégat est traduite ainsi :

1. On trouve le bon environnement abstrait de traduction en utilisant la fonction $\mathbb{F}_a(\mathcal{A}, e^f)$ qui reprend la sémantique de la fonction $\mathbb{F}_e(\mathcal{E}, e^f)$ [15] mais qui s'applique à un environnement abstrait. La sémantique de la fonction $\mathbb{F}_a(\mathcal{A}, e^f)$ est donnée dans la figure IV.13 En parallèle, on utilise la fonction `remove_slices` qui élimine de l'environnement local de NRA^e le même nombre de tranches que le fait $\mathbb{F}_a(\mathcal{A}, e^f)$.
2. On évalue l'expression e^f pour chacun des tuples du paquet courant de l'environnement, avec la même approche que pour la projection ou la sélection avec l'opérateur $\mathtt{push}_{one}$ ($\mathtt{push}_{one}(exp) \ = \ exp \circ^e \{(''slice'' : [\mathtt{In}]), (''tail'' : \mathtt{Env})\}$). Cet opérateur ajoute à l'environnement actuel une tranche contenant un singleton dont l'élément est la valeur courante.

$$\mathcal{T}^{e^a}_{\mathcal{A}}(\mathtt{ag}(e^f)) \quad = \quad \mathcal{T}^{\mathtt{ag}}(\mathtt{ag}, (\overline{\mathcal{T}^{\mathtt{f}}_{(\mathbb{F}_a(\mathcal{A}, e^f))}(\mathtt{push}_{one}(e^f))})) \ \circ^e \ (\mathtt{remove\_slices}\ (\mathcal{A}, e^f))$$

$$\frac{c \in \mathcal{V}}{\mathbb{B}_u(G, c)} \qquad \frac{e^f \notin \mathcal{V} \quad e^f \in G}{\mathbb{B}_u(G, e^f)} \qquad \frac{(\mathtt{fn}(\overline{e^f})) \notin G \quad \bigwedge_{e^f \in \overline{e^f}} \mathbb{B}_u(G, e^f)}{\mathbb{B}_u(G, \mathtt{fn}(\overline{e^f}))}$$

$$\frac{\mathbb{B}_u((A \cup \bigcup_{(A', G) \in \mathcal{A}} G), e^f)}{\mathbb{S}_a(A, \mathcal{A}, e^f)}$$

$$\frac{e^f \notin \mathcal{V} \quad \mathbb{F}_e(\mathcal{E}, e^f) = \mathcal{E}'}{\mathbb{F}_a(((A, G) :: \mathcal{A}), e^f) = \mathcal{E}'}$$

$$\frac{\mathbb{F}_a(\mathcal{A}, e^f) = \mathtt{undefined} \quad \mathbb{S}_a(A, \mathcal{A}, e^f)}{\mathbb{F}_a(((A, G) :: \mathcal{A}), e^f) = (A, G) :: \mathcal{A}}$$

Figure IV.13. *Sémantique de la fonction qui trouve le bon environnement abstrait de traduction d'une expression avec agrégat*

15. Voir III.1.2.5. Sémantique.

IV.5.1.3. Listes d'expressions nommées

L'opérateur de projection de SQL_{Alg} s'applique à une liste d'expressions nommées ($\overline{e_i \texttt{ as } a_i}$). Cette dernière dénote un tuple. Pour traduire une liste d'expressions nommées, nous appliquons la traduction des expressions (simples ou complexes) et la traduction des noms d'attribut :

$$\mathcal{T}^{\text{Sel}}_{\mathcal{A}}(\overline{e_i \texttt{ as } a_i}) \quad = \quad \{\overline{\mathcal{E}^{\text{att}}(a_i) : \mathcal{T}^{e^a}_{A}(e_i)}\}$$

La définition correspondante en Coq est donnée dans la figure IV.14.

```
Fixpoint select_list_to_nraenv nenv s : nraenv :=
  match s with
  | nil ⇒ NRAEnvConst (drec nil)
  | (a,e) :: s ⇒
    NRAEnvBinop
      OpRecConcat
      (NRAEnvUnop (OpRec (attribute_to_string a))
                  (aggterm_to_nraenv nenv e))
      (select_list_to_nraenv nenv s)
  end.
```

Figure IV.14. *Définition en Coq de la traduction des expressions nommées*

IV.5.2. Les formules

La traduction des formules SQL_{Alg} dépend, tout comme les expressions, d'un environnement abstrait. Cette traduction est correct et son théorème de préservation de sémantique est prouvé en Coq.

La définition des formules est paramétrée par des symboles de prédicat génériques. Nous appliquons la même approche que pour les symboles de fonction et d'agrégat en ajoutant une spécification de la traduction des symboles de prédicat.

Les formules sont évaluées dans la logique trivaluée. La formalisation de NRA^e ne supporte pas nativement cette logique. Nous définissons une spécification des opérateurs NRA^e abstraits avec les hypothèses qu'ils préservent la sémantique pour chacun des opérateurs de la logique trivaluée.

IV.5.2.1. Symboles de prédicat

La spécification en Coq de la traduction des symboles de prédicat que nous dénotons par $\mathcal{T}^{\text{pr}}(\ ,\ )$. Comme le montre la figure IV.15, la fonction `predicate_to_nraenv`, prend

```
Class PredicateToNRAEnv
      `{TD : ToData}
      `{BD : BoolBToData (B TRcd)}
: Type :=
  mk_C
    {
      predicate_to_nraenv : predicate → list nraenv → nraenv;
      predicate_to_nraenv_ignores_did : (* ... *);
      predicate_to_nraenv_eq :
        ∀X pred (lx:list X),
        ∀nraenv_of_X value_of_X,
        ∀i dreg,
          (∀x, In x lx →
                ∀did,
                  nraenv_eval i (nraenv_of_X x)  dreg did =
                  Some (value_to_data  (value_of_X x))) →
          ∀did,
            nraenv_eval
              i
              (predicate_to_nraenv pred (map nraenv_of_X lx))
              dreg did =
            Some(boolB_to_data
                   (interp_predicate
                      pred (map value_of_X lx)));
    }.
```

Figure IV.15. *Spécification en Coq de la traduction des symboles de prédicat*

en arguments un symbole de prédicat et une liste d'expressions NRA^e et construit une expression NRA^e. La liste d'expressions correspond à une liste de traductions de formules. La propriété `predicate_to_nraenv_ignores_did` dit que la traduction des symboles de prédicat ne dépend pas de la valeur courante [16]. Une propriété de préservation de sémantique est également admise dans la spécification. Nous reviendrons sur cette propriété dans la section IV.7.

IV.5.2.2. Opérateurs NRA^e de la logique trivaluée

La classe `FormulaToNRAEnv`, présentée dans la figure IV.16, regroupe les différentes spécifications dont nous avons besoin pour la traduction des formules. De plus, nous définissons des opérateurs logiques abstrait, que nous dénotons $\texttt{true}_B, \neg_B, \wedge_B, \vee_B$ dans ce document, mais également les opérateurs abstraits $\texttt{all}_B$, $\texttt{any}_B$, $\texttt{in}_B$ et $\texttt{exists}_B$. Chaque opérateur est accompagnée d'une propriété garantissant qu'il préserve la sémantique.

Ces opérateurs abstraits nous permettent d'avoir une traduction générique et de la définir de manière plus concise. L'encodage des valeurs de la logique trivaluée et les opérateurs abstraits sont définis au niveau de la réalisation.

En utilisant ces briques, nous définissons la traduction (récursive) de formules SQL_{Alg} en NRA^e. Nous commentons ci-dessous les équations qui sont présentées dans figure IV.17.

(1) `true` est traduit par son équivalent syntaxique abstrait $\texttt{true}_B$

$$\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(\texttt{true}) \;=\; \texttt{true}_B$$

(2) , (3) et (4) Il suffit d'appliquer l'opérateur abstrait correspondant aux traductions sous le même environnement abstrait des sous-formules.

$$\begin{array}{lcl} \mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_1 \texttt{ and } f_2) & = & (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_1)) \;\wedge_B\; (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_2)) \\ \mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_1 \texttt{ or } f_2) & = & (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_1)) \;\vee_B\; (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_2)) \\ \mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(\texttt{not } f) & = & \neg_B\; (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f)) \end{array}$$

(5) Nous traduisons un prédicat en lui appliquant l'opérateur abstrait embarquant la sémantique des prédicats et à la liste des traductions sans changer d'environnement abstrait des expressions

$$\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(p\ (\overline{e})) \;=\; \mathcal{T}^{\mathrm{pr}}(p, (\overline{\mathcal{T}^{\mathrm{e}}_{\mathcal{A}}(e)}))$$

(6) , (7) et (8) Pour les opérateurs `any` et `all`, nous utilisons là aussi des opérateurs abstraits après traduction sous le même environnement abstrait de la requête et de

16. Voir III.2.2.3. Sémantique de NRA^e .

```
Class FormulaToNRAEnv
      `{TD : ToData} `{SN : SymbolToNRAEnv TD}
      {EN : EnvToNRAEnv TD} `{FTN : FTermToNRAEnv TD SN EN}
      `{AN : AggregateToNRAEnv TD } `{ATN : ATermToNRAEnv TD SN AN EN FTN
   }
      `{IN : InstanceToNRAEnv TD _ ORN} `{BD : BoolBToData (B TRcd)}
      `{PN :  PredicateToNRAEnv TD BD} : Type :=
  mk_C
    {
      and_or_to_nraenv : and_or → nraenv → nraenv → nraenv;
      and_or_to_nraenv_ignores_did :  (* ... *);
      and_or_to_nraenv_eq :
        ∀a n1 n2, ∀b1 b2 br i dreg,
            (∀did,nraenv_eval br i n1 dreg did =
                   Some (boolB_to_data b1)) →
            (∀did, nraenv_eval br i n2 dreg did =
                    Some (boolB_to_data b2)) →
            ∀did,
              nraenv_eval br i (and_or_to_nraenv a n1 n2) dreg did =
              Some (boolB_to_data   (interp_conj (B TRcd) a b1 b2));
      NRAEnvNotB : nraenv → nraenv;
      NRAEnvNotB_ignores_did : (* ... *);
      NRAEnvNotB_eq : (* ... *);
      NRAEnvIsTrueB : nraenv → nraenv;
      NRAEnvIsTrueB_eq :
        ∀n1,
        ∀b1 br i dreg,
          (∀did,nraenv_eval br i n1 dreg did =
                 Some (boolB_to_data b1)) →
          ∀did,
            nraenv_eval br i (NRAEnvIsTrueB n1) dreg did =
            Some (dbool  (Bool.is_true (B TRcd)  b1));
      NRAEnvSqlExists (nq: nraenv) : nraenv;
      NRAEnvSqlExists_ignores_did : (* ... *);
      NRAEnvSqlExists_eq :(* ... *);
      quantifier_to_nraenv : quantifier →  nraenv → nraenv;
      quantifier_to_nraenv_ignores_did :  (* ... *);
      quantifier_to_nraenv_eq_permut : (* ... *);
      NRAEnvMemberOfB (ls: list String.string) (nsl : nraenv)
                      (lq: list String.string): nraenv → nraenv;
      NRAEnvMemberOfB_ignores_did : (* ... *);
      NRAEnvMemberOfB_eq: (* ... *);
      predicate_to_nraenv_quant_ignores_did : (* ... *);
      predicate_to_nraenv_quant_eq : (* ... *);
    }.
```

Figure IV.16. *Spécification en Coq des opérateurs NRA^e de la logique trivaluée et de la traduction des requêtes SQL_{Alg}*

$$
\begin{array}{lcll}
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(\texttt{true}) & = & \texttt{true}_B & (1)\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_1\ \texttt{and}\ f_2) & = & (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_1))\ \wedge_B\ (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_2)) & (2)\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_1\ \texttt{or}\ f_2) & = & (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_1))\ \vee_B\ (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f_2)) & (3)\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(\texttt{not}\ f) & = & \neg_B\ (\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(f)) & (4)\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(p\ (\overline{e})) & = & \mathcal{T}^{\mathrm{pr}}(p, \overline{(\mathcal{T}^{\mathrm{e}}_{\mathcal{A}}(e))}) & (5)\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(p\ (e, \texttt{all Q})) & = & \texttt{all}_B\ p\ (\mathcal{T}^{\mathrm{e}}_{\mathcal{A}}(e))\ (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{Q})) & (6)\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(p\ (e, \texttt{any Q})) & = & \texttt{any}_B\ p\ (\mathcal{T}^{\mathrm{e}}_{\mathcal{A}}(e))\ (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{Q})) & (7)\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(ls\ \texttt{in Q})) & = & (\mathcal{T}^{\mathrm{Sel}}_{\mathcal{A}}(ls))\ \texttt{in}_B\ (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{Q})) & (8)\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(\texttt{exists Q}) & = & \texttt{exists}_B\ (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{Q})) & (9)
\end{array}
$$

Figure IV.17. *Traduction des formules SQL_{Alg}*

l'expression. Même principe pour l'opérateur `in`. L'opérateur `in` qui apparaît dans le résultat de (8) est un opérateur abstrait à définir dans la réalisation [17].

$$
\begin{array}{lcl}
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(p\ (e, \texttt{all Q})) & = & \texttt{all}_B\ p\ (\mathcal{T}^{\mathrm{e}}_{\mathcal{A}}(e))\ (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{Q}))\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(p\ (e, \texttt{any Q})) & = & \texttt{any}_B\ p\ (\mathcal{T}^{\mathrm{e}}_{\mathcal{A}}(e))\ (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{Q}))\\
\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(ls\ \texttt{in Q})) & = & (\mathcal{T}^{\mathrm{Sel}}_{\mathcal{A}}(ls))\ \texttt{in}_B\ (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{Q}))
\end{array}
$$

(9) Pour l'opérateur `exists`, nous le traduisons par l'opérateur abstrait $\texttt{exists}_B$ que nous implémentons dans la réalisation.

$$\mathcal{T}^{\mathrm{f}}_{\mathcal{A}}(\texttt{exists Q}) = \texttt{exists}_B\ (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{Q}))$$

IV.5.3. Les requêtes

La traduction des requêtes SQL_{Alg} en expressions NRA^e est dénotée par $\mathcal{T}^{\mathrm{Q}}(\ )$. Cette traduction préserve la sémantique. Nous pouvons l'affirmer en nous appuyant la démonstration en Coq d'un théorème de préservation de sémantique [18].

La traduction est correcte pour toute requête SQL_{Alg} bien formée et n'a qu'une seule limitation : on ne peut grouper que sur des noms d'attributs. Nous commentons ci-dessous cette limitation.

— La raison de cette limitation est que dans NRA^e, seul l'opérateur `group_by` permet de grouper selon un critère. Le souci est que cet opérateur est défini uniquement sur des noms d'attributs.

— Le standard ISO de SQL [Melton, 2016] dit que l'on doit grouper uniquement sur des noms d'attribut. Ainsi, la traduction s'applique à toute requête SQL_{Alg} correspondant à une requête SQL respectant ce standard.

17. Il s'agit de `NRAEnvMemberOfB` de la figure IV.16.
18. Voir IV.7. Théorèmes de correction de la traduction.

— Un pré-traitement peut transformer n'importe quelle requête SQL_{Alg} en une requête SQL_{Alg} respectant cette propriété [19]. Ce traitement transformerait, au niveau d'une tranche d'un environnement, le paquet de tuples en lui ajoutant des attributs avec des noms frais et des valeurs correspondant aux évaluations des expressions groupantes pour chacun des tuples et de modifier adéquatement l'ensemble des noms d'attributs et la liste des expressions groupantes de la tranche.

— Il est à noter à ce stade qu'une liste de noms d'attributs est représentable dans le modèle `data`. Il devient donc possible de représenter un environnement de SQL_{Alg} dans une donnée de type `data`. Néanmoins, la partie abstraite n'étant pas nécessaire à l'évaluation des expressions NRA^e que la traduction produit, il nous a paru pertinent de conserver notre approche sur les environnements tel que nous l'avons défini. [20]

Nous décrivons ci-dessous la traduction de chaque type de requête SQL_{Alg}. La traduction complète des requêtes est présentée dans la figure IV.18.

(1) Un nom de table T est traduit puis on lui applique l'opérateur `get` pour dire qu'on veut obtenir la valeur qui lui est associée :

$$\mathcal{T}^{\text{Q}}_{\mathcal{A}}(T) = \texttt{get}(\mathcal{E}^{\text{tab}}(T))$$

(2) , (3) et (4) Pour les opérateurs ensemblistes, il suffit de pousser la fonction de traduction aux sous-requêtes avec le même environnement abstrait et d'appliquer l'opérateur NRA^e équivalent aux traductions des sous-requêtes :

$$\begin{array}{lcl} \mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_1 \texttt{ union } q_2) & = & (\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_1)) \cup (\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_2)) \\ \mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_1 \texttt{ intersect } q_2) & = & (\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_1)) \cap (\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_2)) \\ \mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_1 \texttt{ except } q_2) & = & (\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_1)) \setminus (\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_2)) \end{array}$$

(5) Nous traduisons la jointure SQL_{Alg} par le produit cartésien de NRA^e. Nous pouvons nous permettre cela car les sous-requêtes de SQL_{Alg} ont des noms d'attributs disjoints par construction [21]. On pousse là aussi la traduction aux sous-requêtes en conservant le même environnement abstrait :

$$\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_1 \bowtie q_2) = (\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_1)) \times (\mathcal{T}^{\text{Q}}_{\mathcal{A}}(q_2))$$

(6) Pour la sélection (σ), nous poussons la traduction à la sous-requête pour obtenir la représentation syntaxique en NRA^e des éléments de la sous-requête, puis on lui applique la fonction σ de NRA^e. Le première sous-requête de σ coté cible est $(\texttt{push}_{one}(\mathcal{T}^{\text{f}}_{(\text{sort } q,-)::\mathcal{A}}(f)))$. Pour traduire f, nous ajoutons la tranche $((\text{sort } q, -))$ à l'environnement abstrait. La fonction *sort* permet de récupérer (au niveau syntaxique)

19. La preuve Coq de la correction de ce pré-traitement n'a pas encore été faite.

20. Une des raisons initiale de cette approche était l'impossibilité d'encoder une expression groupante en `data`.

21. Ceci est vrai uniquement pour les requêtes qui sont traduites de SQL_{Coq}, ce qui est toujours le cas dans ce travail.

les noms d'attributs de la table que la requête q dénote. En parallèle, nous ajoutons à chaque fois une tranche à l'environnement local avec $\texttt{push}_{one}$.Comme cet opérateur se situe à l'intérieur de l'opérateur $\sigma_{\langle\rangle}()$, l'élément syntaxique `In` contiendra (itérativement) l'un des éléments de la collection dénotée par $\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q)$.

$$\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\sigma_f(q)) \quad = \quad \sigma_{\left\langle(\texttt{push}_{one}(\mathcal{T}^{\mathrm{f}}_{(\mathrm{sort}\ q,-)::\mathcal{A}}(f)))\right\rangle}\left(\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q)\right)$$

(7) Pour la projection (π), nous adoptons la même mécanique que pour σ à ceci près que nous utilisons l'opérateur $\chi_{\langle\rangle}(\ )$.

$$\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\pi_{\mathrm{ls}}(q)) \quad = \quad \chi_{\left\langle(\texttt{push}_{one}(\mathcal{T}^{\mathrm{Sel}}_{(\mathrm{sort}\ q,-)::\mathcal{A}}(sl)))\right\rangle}\left(\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q)\right)$$

(8) Pour l'opérateur γ. Nous utilisons d'abord le sucre syntaxique $\texttt{group by}_{<a,la>}C$ qui fabrique la partition de C selon les valeurs associée à la et fabrique une collection de tuples. Chaque tuple a un seul attribut a qui contient l'une des partitions. Ensuite, nous construisons une collection de collections en allant les récupérer dans les tuple avec $\cdot a$. Après cela, nous évaluons la formule f pour chaque sous-collection. On peut remarquer que nous ajoutons toute la sous-collection à l'environnement concret en utilisant $\texttt{push}_{bag}$ et que nous ajoutons les expressions groupantes la à l'environnement abstrait $\mathcal{A}$. Pour la projection, nous faisons la même chose.

$$\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\gamma_{(ls,la,f)}(q)) \quad =$$
$$\chi_{\left\langle(\texttt{push}_{bag}(\mathcal{T}^{\mathrm{Sel}}_{(\mathrm{sort}\ q,la)::\mathcal{A}}(sl)))\right\rangle}\left(\sigma_{\left\langle(\texttt{push}_{bag}(\mathcal{T}^{\mathrm{f}}_{(\mathrm{sort}\ q,la)::\mathcal{A}}(f)))\right\rangle}\left(\chi_{\langle\cdot a\rangle}\left(\texttt{group by}_{<a,la>}(\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q))\right)\right)\right)$$

$$\begin{array}{lcll}
\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\mathrm{T}) & = & get(\mathcal{E}^{\mathrm{tab}}(\mathrm{T})) & (1)\\
\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_1\ \texttt{union}\ q_2) & = & (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_1)) \cup (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_2)) & (2)\\
\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_1\ \texttt{intersect}\ q_2) & = & (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_1)) \cap (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_2)) & (3)\\
\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_1\ \texttt{except}\ q_2) & = & (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_1)) \setminus (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_2)) & (4)\\
\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_1 \bowtie q_2) & = & (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_1)) \times (\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q_2)) & (5)\\
\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\sigma_f(q)) & = & \sigma_{\left\langle(\texttt{push}_{one}(\mathcal{T}^{\mathrm{f}}_{(\mathrm{sort}\ q,-)::\mathcal{A}}(f)))\right\rangle}\left(\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q)\right) & (6)\\
\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\pi_{\mathrm{ls}}(q)) & = & \chi_{\left\langle(\texttt{push}_{one}(\mathcal{T}^{\mathrm{Sel}}_{(\mathrm{sort}\ q,-)::\mathcal{A}}(sl)))\right\rangle}\left(\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q)\right) & (7)\\
\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(\gamma_{(ls,la,f)}(q)) & = & & \\
\multicolumn{4}{l}{\chi_{\left\langle(\texttt{push}_{bag}(\mathcal{T}^{\mathrm{Sel}}_{(\mathrm{sort}\ q,la)::\mathcal{A}}(sl)))\right\rangle}\left(\sigma_{\left\langle(\texttt{push}_{bag}(\mathcal{T}^{\mathrm{f}}_{(\mathrm{sort}\ q,la)::\mathcal{A}}(f)))\right\rangle}\left(\chi_{\langle\cdot a\rangle}\left(\texttt{group by}_{<a,la>}(\mathcal{T}^{\mathrm{Q}}_{\mathcal{A}}(q))\right)\right)\right)} & (8)\\
\multicolumn{4}{l}{\text{où } a \text{ est un attribut frais par rapport à } la}
\end{array}$$

Figure IV.18. *Traduction des requêtes SQL_{Alg}*

La spécification montrée en figure IV.19 rassemble toutes les spécifications nécessaires aux différentes traductions qui sont utilisées pour traduire les requêtes $\mathrm{SQL}_{\mathrm{Alg}}$.

```
Class QueryToNRAEnv
      `{TD : ToData} `{SN : SymbolToNRAEnv TD}
      {EN : EnvToNRAEnv TD} `{FTN : FTermToNRAEnv TD SN EN}
      `{AN : AggregateToNRAEnv TD} `{ATN : ATermToNRAEnv TD SN AN EN  FTN
   }
      `{IN : InstanceToNRAEnv TD} `{BD : BoolBToData  (B TRcd)}
      `{PN : PredicateToNRAEnv TD BD}
      `{FN : FormulaToNRAEnv TD SN EN FTN AN ATN IN BD PN}: Type.
```

FIGURE IV.19. *Spécification en Coq de tous les éléments nécessaires à la traduction des requête de SQL$_{Alg}$*

IV.5.4. Exemple de traduction

Nous reprenons une des requêtes SQL$_{\text{Alg}}$ présentées dans III.1.2.3. Exemples de requêtes.

Nous rappelons les schémas des relations pour la requête :

`S : (c:integer)` et `R : (a:integer, b: integer)`

Dan ce qui suit, nous détaillons les étapes de traduction de la requête :

$$\mathcal{T}^{\text{Q}}_{[]}(\pi_{\texttt{(a as a)}}(\sigma_{\texttt{(not (a as a in (}\pi_{\texttt{(c as c)}}S\texttt{)))}}R)) =$$
$$\chi\left\langle(\texttt{push}_{one}(\mathcal{T}^{\text{Sel}}_{[(\{a,b\},-)]}\texttt{(a as a)}))\right\rangle\left(\mathcal{T}^{\text{Q}}_{[]}(\sigma_{\texttt{(not (a as a in (}\pi_{\texttt{(c as c)}}S\texttt{)))}}R)\right)$$
$$\mathcal{T}^{\text{Sel}}_{[(\{a,b\},-)]}\texttt{(a as a)} = \{a : \mathcal{T}^{e^a}_{[(\{a,b\},-)]}(a)\}$$
$$\mathcal{T}^{e^a}_{[(\{a,b\},-)]}(a) = \texttt{first\_of(Env}\cdot\text{"}head\text{"}\texttt{)}\cdot\mathcal{E}^{\text{att}}(a)$$
$$\mathcal{T}^{\text{Q}}_{[]}(\sigma_{\texttt{(not (a as a in (}\pi_{\texttt{(c as c)}}S\texttt{)))}}R) =$$
$$\sigma\left\langle(\texttt{push}_{one}(\mathcal{T}^{\text{f}}_{[(\{a,b\},-)]}\texttt{((not (a as a in (}}\pi_{\texttt{(c as c)}}S\texttt{)))))}\right\rangle\left(\texttt{get}(\mathcal{E}^{\text{tab}}(R))\right)$$
$$\mathcal{T}^{\text{f}}_{[(\{a,b\},-)]}\texttt{((not (a as a in (}\pi_{\texttt{(c as c)}}S\texttt{))))} = \neg_B\,\mathcal{T}^{\text{f}}_{[(\{a,b\},-)]}\texttt{(a as a in (}\pi_{\texttt{(c as c)}}S\texttt{))}$$
$$\mathcal{T}^{\text{f}}_{[(\{a,b\},-)]}\texttt{(a as a in (}\pi_{\texttt{(c as c)}}S\texttt{))} = (\mathcal{T}^{\text{Sel}}_{[(\{a,b\},-)]}\texttt{(a as a)})\ \text{in}_B\ (\mathcal{T}^{\text{Q}}_{[(\{a,b\},-)]}(\pi_{\texttt{(c as c)}}S))$$

$$\mathcal{T}^{\text{Q}}_{[(\{a,b\},-)]}(\pi_{\texttt{(c as c)}}S) = \chi\left\langle(\texttt{push}_{one}(\mathcal{T}^{\text{Sel}}_{[(\{c\},-)::(\{a,b\},-)]}\texttt{(c as c)}))\right\rangle\left(\texttt{get}(\mathcal{E}^{\text{tab}}(S))\right)$$
$$\mathcal{T}^{\text{Sel}}_{[(\{c\},-)::(\{a,b\},-)]}\texttt{(c as c)} = \texttt{first\_of(Env}\cdot\text{"}head\text{"}\texttt{)}\cdot\mathcal{E}^{\text{att}}(c)$$

IV.6. Réalisation des spécifications

Pour que la traduction soit opérationnelle, notamment dans le cadre du projet DBCert que nous présentons dans la section V.1, nous devons définir une instance de chacune des spécifications que nous avons utilisé pour définir les différentes traductions. Ces instances

utilisent les différents éléments définis dans la réalisation [22] des éléments génériques de SQL_{Alg}. La figure IV.20 présente les différents types de valeurs et les différents symboles que nous devons encoder ou traduire. Nous exposons dans cette section les différentes réalisations. La traduction des noms d'attribut étant triviale, nous l'omettons dans cette section.

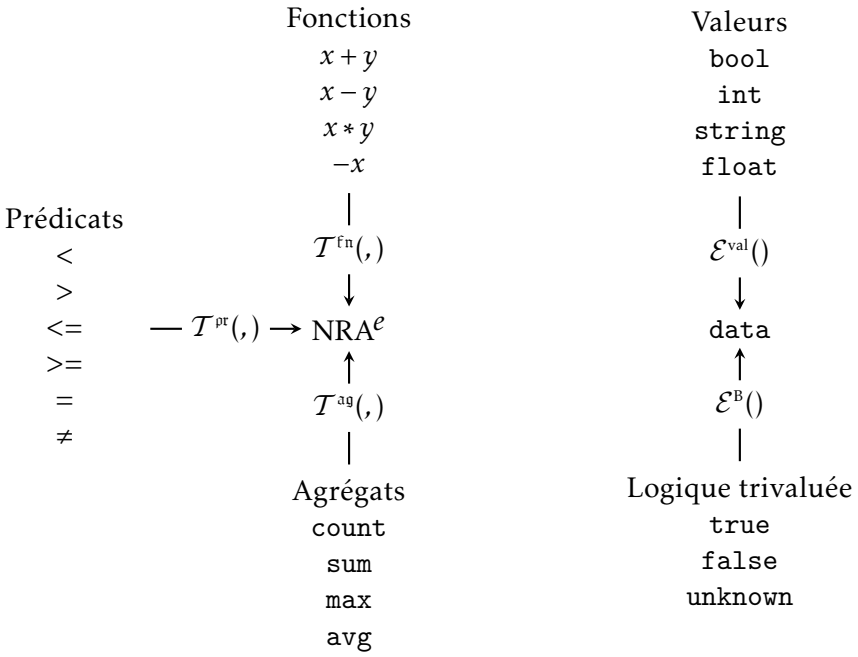


Figure IV.20. *Représentation schématique des instances*

IV.6.1. Les valeurs nulles

Le modèle de NRA^{e} ne supporte pas de manière native les valeurs nulles. Afin de les définir, nous utilisons le type de données `either` pour les encoder. Dans un certain sens, une valeur nulle est typée, autrement dit, une valeur nulle de type entier est différenciée d'une valeur nulle booléenne. A cet effet, pour encoder la valeur nulle de type entier par exemple, nous le représentons dans le modèle cible par (`right` 0). Pour la valeur nulle de type `string`, nous utilisons la valeur (`right ""`). Enfin, pour toute valeur non nulle v, nous la représentons par (`left` v). Nous pouvons voir sur la figure IV.21 la définition en Coq de l'encodage des valeurs, la preuve que cet encodage est une injection et la définition de l'instance de la spécification relative au modèle de données.

22. Voir III.1.2.6. Preuve de concept : Une instance de la spécification.

```
Definition value_to_data : value → data.
  intros [[s | ] | [z | ] | [b | ] | [f | ]];
    [exact (dleft (dstring s))|exact (dright (dstring ""))
     |exact (dleft (dnat z))|exact (dright (dnat 0))
     |exact (dleft (dbool b))| exact (dright (dbool true))
     |exact (dleft (dfloat f))| exact (dright (dfloat float_zero))].
Defined.

Lemma value_to_data_inj :
  ∀a1 a2, value_to_data a1 = value_to_data a2 → a1 = a2.
Proof.
  intros [[s1| ]|[z1| ]|[b1| ]|[f1| ]]
         [[s2| ]|[z2| ]|[b2| ]|[f2| ]] H1;
    inversion H1;apply eq_refl.
Qed.

Global Instance tnull_TD : ToData :=
  ToData.mk_C TNull attribute_to_string
              attribute_to_string_compare_lt
              value_to_data  value_to_data_inj.
```

FIGURE IV.21. *Réalisation de l'encodage des valeurs*

Exemples d'encodage de valeurs

42	`left 42`	"Doe"	`left "Doe"`
$\texttt{NULL}_{\texttt{int}}$	`right 0`	$\texttt{NULL}_{\texttt{string}}$	`right ""`

IV.6.2. Valeurs de la logique trivaluée

Concernant la logique trivaluée, nous adoptons la même mécanique que celle appliquée aux valeurs. Nous utilisons (`left true`), (`left false`) et (`right unit`) pour représenter respectivement `true`, `false` et `unknown`. Les définitions et théorèmes Coq correspondants sont présentés sur la FIGURE IV.22.

Illustration de l'encodage des valeurs de la logique trivaluée

$\texttt{true}_3$	`left true`
$\texttt{false}_3$	`left false`
`unknown`	`right unit`

```
Definition bool3_to_data b : data :=
  match b with
  | Bool3.true3 ⇒ dleft (dbool true)
  | Bool3.false3 ⇒ dleft (dbool false)
  | Bool3.unknown3 ⇒ dright dunit
  end.

Lemma bool3_to_data_true :
  bool3_to_data (Bool.true Bool3.Bool3) =
  dleft (dbool true).
Proof.
  apply eq_refl.
Qed.

 (* ... *)
Global Instance tnull_BD : BoolBToData (B TRcd) :=
  BoolBToData.mk_C Bool3 (dleft (dbool true))
                   bool3_to_data bool3_to_data_true (* ... *).
```

Figure IV.22. *Réalisation de l'encodage de la logique trivaluée*

IV.6.3. Les symboles de fonction

Dans la réalisation de SQL_{Alg} que nous utilisons pour instancier nos spécifications, et dans SQL en général, les valeurs nulles sont des éléments absorbants des symboles de fonctions binaires. Pour cette raison, même si ces symboles ont des opérateurs équivalents dans NRA^e, nous devons isoler les cas où au moins l'un des deux arguments d'un symbole de fonction binaire est nul du cas où aucun des deux arguments ne l'est. Nous le faisons en deux étapes.

La première étape est la définition d'un opérateur de composition absorbant à droite. Nous le dénotons par $\circ_\star$ et nous le définissons par :

$$n_1 \circ_\star n_2 \quad = \quad (n_1 \oplus \texttt{right}\,(\texttt{In})) \circ n_2$$

L'opérateur $\circ_\star$ observe si le résultat de l'évaluation de son deuxième opérande (n_2) est une valeur nulle, et dans ce cas renvoie ce de cette évaluation (en reconstruisant avec un `right`). En revanche, si l'évaluation de n_2 n'est pas nulle, l'opérateur renvoie l'évaluation de n_1. Nous pouvons donner une définition de cet opérateur qui offre une meilleure intuition : $n_1 \circ_\star n_2 = (n_1 \oplus n_2) \circ n_2$. Mais cette définition n'est pas optimale puisque elle engendre, lorsque le résultat de l'évaluation de n_2 est nulle, une deuxième évaluation de n_2 qui, notons le, est la même que l'évaluation de (`right In`) dans ce contexte.

Dans la deuxième étape, qui s'appuie sur l'opérateur $\circ_\star$, nous définissons, pour chaque opérateur natif de NRA^e qui correspond à l'un des symboles de fonction de SQL_{Alg}, une

version où notre encodage des valeurs nulles permet à ces dernières d'être absorbantes. Pour un opérateur $\otimes$, la définition est :

$$n_1 \otimes_\star n_2 \quad = \quad (((\texttt{left}\,(\texttt{In} \otimes (\texttt{In} \circ_\star n_2))) \circ_\star n_1) \circ_\star n_2) \circ_\star n_1$$

Comme l'opérateur $\circ_\star$ est absorbant à droite, lisons cette définition en commençant de la droite. Si n_1 dénote une valeur nulle, alors on renvoie cette dernière. Sinon, si n_2 dénote une valeur nulle, alors on renvoie celle-ci. Sinon, nous sommes dans le cas où aucun des deux opérandes ne dénote une valeur nulle. Dans ce cas, qui correspond à l'expression $((\texttt{left}\,(\texttt{In} \otimes (\texttt{In} \circ_\star n_2))) \circ_\star n_1)$, la valeur que dénote n_2 est de la forme $\texttt{left}\ v_2$ et la valeur courante (de retour) est v_2. Afin d'évaluer le résultat final, nous devons récupérer la valeur v_1 sachant que n_1 est de ma forme $\texttt{left}\ v_1$, ce qui est fait dans cette expression. Nous avons défini cet opérateur de plusieurs manières avant d'arriver à cette définition adaptée aux preuves. Une des perspectives de ce travail est de définir ces opérateurs intermédiaires de manière plus optimale et de prouver leurs équivalences à ceux utilisés pour les preuves.

Nous présentons ci-dessous l'opérateur d'addition défini avec la méthode décrite précédemment.

$$\begin{aligned}
[\![\texttt{left 2} +_\star \texttt{left 1}]\!]^{\text{NRA}^e}_{<db,\gamma,d>} &= [\![(((\texttt{left}\,(\texttt{In} + (\texttt{In} \circ_\star \texttt{left 1}))) \circ_\star \texttt{left 2}) \circ_\star \texttt{left 1}) \circ_\star \texttt{left 2}]\!]^{\text{NRA}^e}_{<db,\gamma,d>} \\
&= [\![((\texttt{left}\,(\texttt{In} + (\texttt{In} \circ_\star \texttt{left 1}))) \circ_\star \texttt{left 2}) \circ_\star \texttt{left 1}]\!]^{\text{NRA}^e}_{<db,\gamma,2>} \\
&= [\![(\texttt{left}\,(\texttt{In} + (\texttt{In} \circ_\star \texttt{left 1}))) \circ_\star \texttt{left 2}]\!]^{\text{NRA}^e}_{<db,\gamma,1>} \\
&= [\![\texttt{left}\,(\texttt{In} + (\texttt{In} \circ_\star \texttt{left 1}))]\!]^{\text{NRA}^e}_{<db,\gamma,2>} \\
&= \texttt{left}\,(2 + [\![(\texttt{In} \circ_\star \texttt{left 1})]\!]^{\text{NRA}^e}_{<db,\gamma,2>}) \\
&= \texttt{left}\,(2 + [\![\texttt{In}]\!]^{\text{NRA}^e}_{<db,\gamma,1>}) \\
&= \texttt{left}\,(2 + 1)
\end{aligned}$$

IV.6.4. Les symboles d'agrégat

Les valeurs nulles sont des éléments neutres des symboles d'agrégat (hormis le symbole `count`). Une des solutions est de filtrer la liste d'arguments que prend le symbole et de ne garder que les valeurs non nulles, puis d'appliquer le symbole uniquement à la liste résultat :

$$\texttt{remove\_nulls}\ n = \sigma_{\langle \texttt{true} \oplus \texttt{false} \rangle}(n)$$

Une fois cet que opérateur est défini, la traduction des symboles d'agrégat est implémentée en l'utilisant et en appliquant un opérateur NRA^e avec une sémantique équivalente à la liste filtrée.

IV.6.5. Les symboles de prédicat et les opérateurs logiques

Afin de définir une instance de la spécification de la traduction des formules, nous proposons des opérateurs NRA^e ayant une sémantique équivalente à celle des opérateurs

de la logique trivaluée. Nous les représentons par $\neg_B$, $\wedge_B$ et $\vee_B$, en voici les définitions :

— Pour une expression NRA^e n qui dénote une valeur de la logique trivaluée (selon l'encodage de cette réalisation), la négation est définie par :

$$\neg_B\ n = (\texttt{left}\ (\neg\texttt{In})\ \oplus\ \texttt{right unit}) \circ n$$

Cet opérateur prend l'évaluation de l'expression NRA^e n comme valeur courante ($\circ$). Si cette dernière est égale à une valeur de la forme (`right _`), l'opérateur renvoie `right unit`, sinon (si la valeur de n est de la forme (`left _`)), l'opérateur renvoie la négation de la valeur de n qui est forcément (`left true`) ou (`left false`).

— L'opérateur abstrait $\wedge_B$ est défini en utilisant l'opérateur intermédiaire `hd_def` qui s'applique à une donnée d et à une expression n qui dénote une collection et qui renvoie le premier élément de la (liste qui implémente la) collection s'il existe ou une valeur par défaut (si la collection est vide) :

$$\texttt{hd\_def}\ d\ n\ \ =\ \ (\texttt{In}\ \oplus\ d)\ \circ (\ \texttt{head}\ n)$$

Ensuite, des mécanismes similaires à celui des symboles de fonctions et celui de la négation sont usités :

$$\begin{aligned} n_1\ \wedge_B\ n_2\ \ =\ \ & (((((\texttt{left}\ (\texttt{In} \wedge (\texttt{In} \circ_\star n_2))) \circ_\star n_1)\ \oplus\ \texttt{hd\_def}\ (\texttt{right unit})\ (\sigma_{\langle \neg\texttt{In} \oplus \texttt{false}\rangle}([n_1])))\\ & \quad \circ n_2)\\ & \ \oplus\\ & \ ((\texttt{hd\_def}\ (\texttt{right unit})\ (\sigma_{\langle \neg\texttt{In} \oplus \texttt{false}\rangle}([n_2]))\ \oplus\ \texttt{right unit}) \circ n_2))\\ & \circ\\ & n_1 \end{aligned}$$

— Le principe de la définition de l'opérateur $\vee_B$ est proche de celui de $\wedge_B$:

$$\begin{aligned} n_1\ \vee_B\ n_2\ \ =\ \ & (((((\texttt{left}\ (\texttt{In} \vee (\texttt{In} \circ_\star n_2))) \circ_\star n_1)\ \oplus\ \texttt{hd\_def}\ (\texttt{right unit})\ (\sigma_{\langle \texttt{In} \oplus \texttt{false}\rangle}([n_1])))\\ & \quad \circ n_2)\\ & \ \oplus\\ & \ ((\texttt{hd\_def}\ (\texttt{right unit})\ (\sigma_{\langle \texttt{In} \oplus \texttt{false}\rangle}([n_2]))\ \oplus\ \texttt{right unit}) \circ n_2))\\ & \circ\\ & n_1 \end{aligned}$$

Nous définissons aussi l'opérateur $\texttt{is\_true}_B$ qui permet de déconstruire les valeurs de la logique trivaluée vers la logique native de NRA^e. Cet opérateur est utilisé en association avec l'opérateur $\sigma_{\langle\ \rangle}(\ )$ qui supporte la logique à deux valeurs.

$$\texttt{is\_true}_B\ n = (\texttt{In}\ \oplus\ \texttt{false}) \circ n$$

Cet opérateur permet de traduire une expression qui dénote une valeur (`left true`) vers la valeur (`true`), et une expression qui dénote une des deux valeurs (`left true`) et (`left true`) vers la valeur (`false`).

Pour l'opérateur abstrait $\texttt{exists}_B$, en utilisant les opérateurs unaires natifs des NRA^e : $\neg$ et count, et l'opérateur binaire $\leq$, nous définissons l'opérateur intermédiaire not_empty qui s'appliqueà une expression dénotant une collection. Cet opérateur renvoie false lorsque la collection est vide et true lorsque la collection contient au moins un élément :

$$\texttt{is\_not\_empty}\ n = \neg(\texttt{count}\ n \leq 0)$$

Ensuite, nous passons des booléens natifs vers notre représentation de la logique trivaluée avec le constructeur left :

$$\texttt{exists}_B\ n = \texttt{left}\ (\texttt{is\_not\_empty}\ n)$$

Enfin, pour définir l'opérateur $\texttt{in}_B$ qui manipule en même temps la logique à trois valeurs et les valeurs nulles, nous avons d'abord défini l'opérateur intermédiaire if qui prend en argument trois expressions NRA^e. La première s'évalue en booléen natif. Si sa valeur est true, on évalue le deuxième argument, si c'est false, on évalue le troisième :

$$\texttt{if}\ cond\ nyes\ nno = (nyes \oplus nno) \circ (\text{head}\ (\sigma_{\langle \texttt{In} \rangle}([ncond])))$$

On définit également l'opérateur rec_contains_null qui renvoie vrai ou faux selon que l'enregistrement dénoté par l'expression à laquelle l'opérateur est appliqué est appliqué contient une valeur nulle ou non.

Enfin, en utilisant un certain nombre d'opérateurs intermédiaires que nous avons présenté dans cette section, nous définissons l'opérateur $\texttt{in}_B$. Cette définition, présentée dans la figure IV.23, permet d'effectuer la preuve de préservation de sémantique car elle reflète la manière dont l'opérateur in de SQL_{Alg} est évalué.

Pour chacun de tous ces opérateurs, nous avons prouvé la préservation de la sémantique pour pouvoir construire l'instance de la spécification relative à la traduction des formules.

IV.7. Théorèmes de correction de la traduction

La traduction de SQL_{Alg} vers NRA^e est certifiée correcte en Coq en terme de préservation de sémantique. la figure IV.24 expose la diagramme de simulation de la traduction. Le théorème 4.1, prouvé en Coq, garantit la préservation de la sémantique.

Théorème 4.1
Pour toute instance de base de données de SQL_{Alg} i et toute valeur courante de NRA^e d, l'évaluation de la traduction de toute requête q en expression NRA^e, en encodant l'instance i, est égale, modulo permutations, à l'évaluation de cette requête encodée en collection de type data.

$$\forall i\ d\ q, [[\mathcal{T}^Q(q)]]^{NRA^e}_{<\mathcal{E}^{db}(i),[],d>} =_{bags} \mathcal{E}^{bag}([[q]]^q_{<i,[]>}) \qquad \square$$

```
in_B r n  =  if (is_not_empty n)
               (if (rec_contains_null r)
                   (right unit)
                   (if (r ∈ n)
                       (left true)
                       (if (is_not_empty (σ_⟨rec_contains_null In⟩(n)))
                           (right unit)
                           (left false))
                   )
               )
               (left false)
```

Figure IV.23. *Définition de l'opérateur* $\texttt{in}_B$

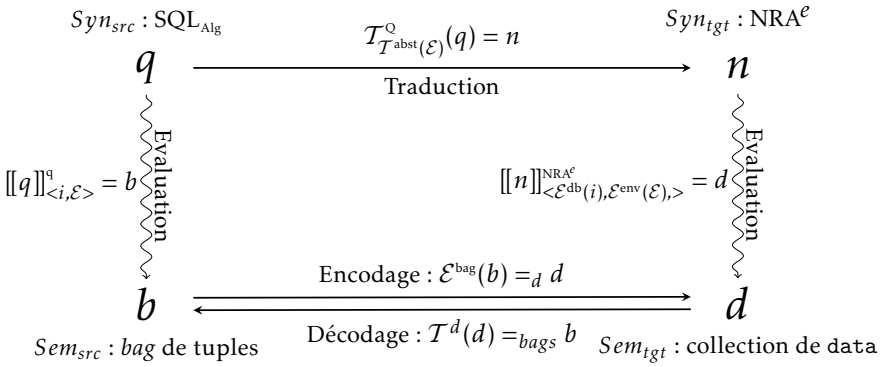


Figure IV.24. *Diagramme de simulation de la traduction*

L'évaluation d'une requête se fait sous un environnement initial vide. Si la requête contient une sous-requête, cette dernière est récursivement évaluée sous un environnement auquel on a ajouté une tranche. Afin de prouver le théorème 4.1, il est nécessaire de démontrer le théorème 4.2 en généralisant l'environnement d'évaluation.

Théorème 4.2
Pour toute instance de base de données de SQL_{Alg} i, tout environnement de SQL_{Alg} $\mathcal{E}$ et toute valeur courante de NRA^e d, l'évaluation de la traduction de toute requête q en expression NRA^e, en encodant l'instance i et les parties concrète et abstraite de $\mathcal{E}$, est égale, modulo permutations, à l'évaluation de cette requête encodée en collection de type `data`.

$$\forall i\ \mathcal{E}\ d\ q, [[\mathcal{T}^{Q}_{\mathcal{T}^{abst}(\mathcal{E})}(q)]]^{NRA^e}_{<\mathcal{E}^{db}(i),\mathcal{E}^{env}(\mathcal{E}),d>} =_{bags} \mathcal{E}^{bag}([[q]]^{q}_{<i,\mathcal{E}>}) \qquad \square$$

En appliquant le même raisonnement aux (sous-)formules et aux (sous-)expressions, nous avons également prouvé en Coq le théorème 4.3 pour les formules, le théorème 4.4 et le théorème 4.5 pour les expressions.

Théorème 4.3

$$\forall i\ \mathcal{E}\ d\ f, [[\mathcal{T}^{f}_{\mathcal{T}^{abst}(\mathcal{E})}(f)]]^{NRA^e}_{<\mathcal{E}^{db}(i),\mathcal{E}^{env}(\mathcal{E}),d>} = \mathcal{E}^{B}([[f]]^{f}_{\mathcal{E}}) \qquad \square$$

Théorème 4.4

$$\forall \mathcal{E}\ d\ db\ e^a, [[\mathcal{T}^{e^a}_{\mathcal{T}^{abst}(\mathcal{E})}(e^a)]]^{NRA^e}_{<db,\mathcal{E}^{env}(\mathcal{E}),d>} = \mathcal{E}^{val}([[e^a]]^{a}_{\mathcal{E}}) \qquad \square$$

Théorème 4.5

$$\forall \mathcal{E}\ d\ db\ e^f, [[\mathcal{T}^{e^f}_{\mathcal{T}^{abst}(\mathcal{E})}(e^f)]]^{NRA^e}_{<db,\mathcal{E}^{env}(\mathcal{E}),d>} = \mathcal{E}^{val}([[e^f]]^{e}_{\mathcal{E}}) \qquad \square$$

Comme expliqué dans la section II.3, plusieurs visions de la correction des compilateurs existent. Ici, nous utilisons l'une des visions les plus exigeantes [Mosses, 1980]. Ceci est possible, car l'encodage est un isomorphisme et le décodage est isomorphique si on se limite aux valeurs que peut produire l'encodage. Nous présentons pour finir, le théorème 4.6, qu'on peut déduire à partir du théorème 4.2 et de l'injectivité du décodage.

Théorème 4.6

$$\forall i\ \mathcal{E}\ d\ q, \mathcal{T}^{d}([[\mathcal{T}^{Q}_{\mathcal{T}^{abst}(\mathcal{E})}(q)]]^{NRA^e}_{<\mathcal{E}^{db}(i),\mathcal{E}^{env}(\mathcal{E}),d>}) =_{bags} [[q]]^{q}_{<i,\mathcal{E}>} \qquad \square$$

IV.8. Conclusion

Nous avons présenté dans ce chapitre le travail de recherche de notre thèse dans ses aspects formels et théorique. Dans le chapitre qui suit, nous discutons ce travail dans le cadre plus opérationnel et plus applicatif du projet DBCert.

Troisième partie : Discussion

La traduction certifiée en Coq proposée dans cette thèse contribue de manière décisive dans la construction du compilateur certifié DBCert. Ce compilateur traduit des requêtes SQL vers du code impératif. DBCert a le potentiel de devenir plate-forme d'interopérabilité de langages centrés données permettant de formaliser et d'étudier finement cette famille de langages tout en offrant une solution de compilation produisant du code opérationnel.

Dans ce chapitre, nous discutons notre travail de recherche dans le cadre opérationnel et applicatif du projet DBCert. Nous montrons l'importance de notre contribution dans ce travail plus global. Nous proposons un exemple simple d'utilisation du compilateur de DBCert. Ensuite, nous confrontons DBCert à d'autre outils comparables. Enfin, nous situons notre travail en décrivant quelques travaux de recherche sur la formalisation des langages centrés données.

Chapitre

V

Discussion

V.1 Cas d'application : DBCert **105**
V.1.1 Présentation de DBCert 105
V.1.2 Préservation de la sémantique 106
V.1.3 Exemple 106
V.1.4 Évaluation de DBCert 107
V.1.4.1 AlaSQL 108
V.1.4.2 *Benchmarks* 108
V.1.4.3 Résultats 109
V.1.4.4 Performances 110
V.2 Travaux connexes **110**

V.1. Cas d'application : DBCert

V.1.1. Présentation de DBCert

Le travail réalisé pour la préparation de notre thèse contribue à un projet né d'une collaboration entre V. Benzaken [†], É. Contejean [††], M. Hachmaoui [†] et C. Keller [†] d'une part et L. Mandel [‡], A. Shinnar [‡] et J. Siméon [‡‡] de l'autre. Ce projet est baptisé DBCert. DBCert est un compilateur certifié en Coq qui traduit les requêtes SQL du fragment `select from where group by having` vers du code impératif. Ce code impératif est compilé vers du JavaScript, mais d'autres *backends* peuvent être ajoutés. La traduction certifiée présentée dans le chapitre IV a permis d'établir une connexion entre SqlCert et Q*Cert.

La figure V.1 présente l'architecture de la chaîne de compilation DBCert. Les langages qui se trouvent dans une boite grise sont mécanisés en Coq et les étapes de compilation représentées par des arcs gris sont certifiées correctes. On constate que toute la partie qui

†. Université Paris-Saclay
††. CNRS - Université Paris-Saclay
‡. IBM Research
‡‡. Clause Labs

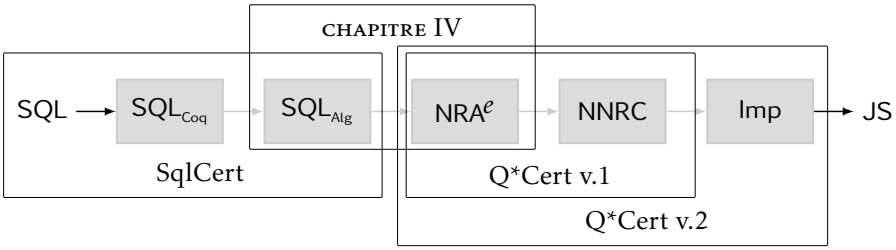


Figure V.1. *Architecture du projet DBCert*

va de SQL$_{Coq}$ à Imp est complètement certifiée. En amont de la chaîne, nous utilisons un parseur non certifié qui transforme la requête SQL en entrée en requête SQL$_{Coq}$. En aval, nous utilisons également un générateur de code non certifié qui transforme le code Imp en code JavaScript. Ces étapes initiale et finale font de DBCert un compilateur réaliste et utilisable.

Le langage Imp a été conçu spécifiquement pour DBCert. C'est une formalisation en Coq d'un langage impératif générique simple. Par générique, on veut dire que le langage est paramétré par un modèle de données abstrait, des opérateurs natifs abstraits et un contexte d'exécution abstrait. Cette généricité est dans l'esprit de celle de SqlCert et celle de la compilation issue de notre thèse. Elle permet à Imp de représenter toute une famille de langages impératifs simples en instanciant ses paramètres de manière adaptée à tel ou tel langage. La sémantique de Imp est très simple, ce qui permet de générer du code dans un langage cible, avec confiance, même sans certification.

V.1.2. Préservation de la sémantique

En utilisant les théorèmes de préservation de sémantique de chaque compilateur de la chaîne, on arrive à prouver le théorème de correction global théorème 5.1.

Théorème 5.1 (Préservation de la sémantique)
Pour un schéma i et une requête SQL$_{Coq}$ Q donnés :

— **si** *Q est bien formée, dans le sens de [Benzaken et Contejean, 2019],*

— **alors** *le compilateur produit une requête q tel que :*

$$[[Q]]^{q}_{<i,[]>} \text{ est égale (égalité multiensembliste) à } [[q]]^{imp}(\mathcal{T}^{i}(i))$$

□

V.1.3. Exemple

On présente ici un exemple de compilation et d'exécution d'une requête SQL simple.

Prenons un fichier `tests/org2.sql` qui contient une requête `select from where` simple. Il contient également une requête `create table` qui nous permet de connaître le schéma de la table, car cette information est utilisée dans la compilation.

```
> cat tests/org2.sql
create table books (lid int, title text);
select name from books where lid < 3;
```

La commande `./dbcert -link` permet de compiler la requête SQL et d'écrire le code obtenu dans `tests/org2.js`.

```
> ./dbcert -link tests/org2.sql
Corresponding JS query generated in: tests/org2.js
Compilation to JavaScript finished
```

A présent, on a besoin d'un fichier JSON qui a le même schéma que celui spécifié dans le fichier d'origine, par exemple, `tests/db1.json`.

```
> cat tests/db1.json
{
  "books": [
    { "lid" : 1, "title" : "Foundations of databases" },
    { "lid" : 2, "title" : "Software foundations" },
    { "lid" : 3, "title" : "Database management systems" },
    { "lid" : 4, "title" : "Types and programming languages" },
    { "lid" : 5, "title" : null }
  ]
}
```

Enfin, la commande `node ./dbcertRun.js` appliquée à un code JavaScript produit par DBCert (`tests/org2.js`) et à un fichier JSON avec un schéma adéquat produit un résultat correct.

```
> node ./dbcertRun.js tests/org2.js tests/db1.json
[{"title":"Foundations of databases"},{"title":"Software foundations"}]
```

V.1.4. Évaluation de DBCert

Afin d'évaluer DBCert, nous l'avons comparé à deux autres compilateurs : Q*Cert qui contenait déjà une ancienne implémentation de SQL et AlaSQL que nous allons présenter.

V.1.4.1. AlaSQL

AlaSQL est une implémentation de SQL pour Node.js qui permet d'interroger une base de données plate dans un fichier JSON. Ci-dessous un exemple basique de code AlaSQL.

```
alasql("create table books (lid int, title text)");
alasql.tables.R.data = [
    { "lid" : 1, "title" : "Foundations of databases" },
    { "lid" : 2, "title" : "Software foundations" },
    { "lid" : 3, "title" : "Database management systems" },
    { "lid" : 4, "title" : "Types and programming languages" },
    { "lid" : 5, "title" : null }];
var res = alasql("select name from books where lid < 3");
-- res = [{"title":"Foundations of databases"},{"title":"Software
    foundations"}]
```

Dans la première ligne, on définit une table `books` à deux colonnes `lid` et `title`. La deuxième ligne remplit la table avec 5 tuples. Enfin, la dernière ligne exécute une requête `select from where` et renvoie le résultat correspondant.

V.1.4.2. *Benchmarks*

Afin de se convaincre de la pertinence du besoin de certification qui nous a poussé à concevoir DBCert, nous avons utilisé des *benchmarks* [1] sémantiques issus de [Benzaken *et al.*, 2014, Guagliardo et Libkin, 2017].

Les benchmarks insistent sur les deux points les plus subtiles dans la sémantique de SQL qui sont les valeurs nulles et les requêtes corrélées.

Nous listons ci-dessous quelques unes de ces requêtes :

— Valeurs nulles :

```
create table R (A double precision);
create table S (A double precision);
create table T (A double precision);

select R.A from R where R.A not in (select S.A from S);
select R.A from R where not exists (select * from S where S.A = R.A);
select R.A from R except select S.A from S;
select T.A,count(*) as c from T group by T.A;
```

— Requêtes corrélées :

1. Un total de 15 requêtes couvrant des points très particuliers de la sémantique de SQL.

```
create table t1 (a1 double precision, b1 double precision);
create table t2 (a2 double precision, b2 double precision);

select a1, max(b1) from t1 group by a1;

select a1 from t1 group by a1
having exists
        (select a2 from t2 group by a2 having sum(1.0+0.0*a1) = 10.0);

select a1 from t1 group by a1
having exists
        (select a2 from t2 group by a2 having sum(1.0+0.0*a2) = 10.0);

select a1 from t1 group by a1
having exists
  (select a2 from t2 group by a2 having sum(1.0+0.0*a2) = 2.0);

select a1 from t1 group by a1
having exists
  (select a2 from t2 group by a2 having sum(1.0+0.0*a1+0.0*b2) = 3.0);
-- ...
```

La sémantique de référence respecte le standard SQL. Lorsqu'elle est trop subtile, c'est les résultats des trois *SGBD* relationnels les plus utilisés [2] qui font foi.

V.1.4.3. Résultats

La table V.1 résume les taux de résultats valides pour chacun des trois compilateurs. Cette comparaison montre que seul DBCert couvre la totalité du fragment sémantique des benchmarks. Il s'est avéré que l'ancienne implémentation de SQL dans Q*Cert est incomplète et qu'elle ne supporte pas les valeurs nulles. On constate également que AlaSQL ne donne pas toujours un résultat correct, que ce soit pour les valeurs nulles ou pour les requêtes corrélées.

Benchmarks	DBCert	AlaSQL	Q*Cert
Valeurs nulles	100%	75%	N/A
Requêtes corrélées	100%	64%	82%

Table V.1. *Taux de résultats valides sur les requêtes des benchmarks sémantiques*

2. Oracle, PostgreSQL et SQLite.

V.1.4.4. Performances

En terme de performance, DBCert donne des résultats raisonnables et satisfaisants. Il comporte plusieurs optimisations, pendant la traduction, dans le moteur d'exécution ou par réécriture. Cependant, pour les requêtes complexes avec agrégats lorsque la base de données est très grande, DBCert devient moins performant mais reste raisonnablement utilisable. Par exemple, pour une base de données qui contient 58000 tuples et une requête avec la clause `group by` et des symboles d'agrégat, DBCert est 2 fois moins rapide que AlaSQL.

V.2. Travaux connexes

Afin de mieux situer notre travail de thèse, nous abordons dans cette section les travaux de formalisation des langages centrés données. Curieusement, les méthodes formelles n'ont été que peu utilisées pour formaliser et étudier les langages et systèmes centrés données.

Les deux projets [Benzaken et Contejean, 2019] et [Auerbach *et al.*, 2017b], entre lesquels notre thèse jette un pont, font partie des travaux de recherches les plus aboutis qui appliquent les méthodes formelles aux langages centrés données. Comme nous les avons présentés dans le chapitre III, il est inutile de les présenter ici. Il est à noter que Q*Cert dispose d'une formalisation native de SQL. Celle-ci ne gère que les corrélations entre les requêtes successives. En ne supporte pas nativement les valeurs nulles telles que définies dans SQL, ni la logique trivaluée.

Les premiers travaux de recherche à s'être intéressés à la thématique ont naturellement étudié l'algèbre relationnelle. La première formalisation [Gonzalia, 2006, Gonzalia, 2003] de l'algèbre relationnelle a été effectuée en Agda [The Agda Development Team, 2010]. En 2014, une formalisation complète en Coq de l'algèbre relationnelle [Benzaken *et al.*, 2014] est proposée.

Le langage SQL a également fait l'objet de plusieurs formalisations. Outre SqlCert, le travail présenté dans [Malecha *et al.*, 2010] introduit une formalisation et une vérification d'un fragment limité de SQL avec une représentation des attributs par position. Ce fragment ne contient ni les clauses `group by` et `having`, ni les quantificateurs, ni les valeurs nulles, ni les agrégats. Enfin, elle ne permet pas l'imbrication de requête qui implique une gestion sophistiquée des environnements.

En 2017, l'outil HottSQL [Chu *et al.*, 2017] qui vérifie si deux requêtes SQL sont équivalentes est introduit. Comparé à [Malecha *et al.*, 2010], HottSQL supporte les agrégats. Ces deux formalisations ne produisent pas du code exécutables et ne permettent donc pas de construire des compilateurs certifiés opérationnels.

Un des premier travaux a s'être penché sur la problématique liée à la à l'imbrication des requêtes de SQL est communiqué dans [Kim, 1982]. Son auteur y propose un algo-

rithme de désimbrication de requêtes par réécriture d'une version minimale de SQL. Ce travail pointe un enjeu important autour du langage SQL mais est très limité en terme de fragment considéré. Dans la même lignée, [Ganski et Wong, 1987, Dayal, , Seshadri *et al.*, 1996] ont tous proposé de nouvelles solutions pour décorréler les requêtes SQL. Tous ces travaux proposent des solutions basées sur des algorithmes de réécriture et ne sont pas certifiées.

Dans [Cao et Badia, 2007], les auteurs proposent une optimisation des requêtes corrélées de SQL en les traduisant vers une algèbre relationnelle imbriquée. Même si ce travail gère correctement les valeurs nulles, il se limite au fragment `select from where` et ne supporte pas les agrégats. De plus, la proposition n'est pas implémentée ni certifiée correcte.

D'autres langages centrés données ont également connu des travaux de formalisation. En 2004, le langage XPath a été formalisé en Coq [Genevès et Vion-Dury, 2004]. En 2010, un fragment du langage XQuery a été spécifié en Isabelle[Cheney et Urban, 2011]. Un moteur d'exécution[Benzaken *et al.*, 2017] mécanisé en Coq du langage Datalog a aussi été formalisé.

Dans [Auerbach *et al.*, 2017a] et [Auerbach *et al.*, 2017c], les auteurs font état de leur expérience dans la conception de Q*Cert et plus généralement du prototypage d'un compilateur de requêtes certifié.

Enfin, plus récemment, une formalisation mécanisée en Coq du langage GraphQL a été proposée dans [Díaz *et al.*, 2020].

Conclusion et perspectives

Nous avons introduit dans ce document une traduction d'une algèbre relationnelle étendue (SQL_{Alg}) qui capture toute la sémantique du fragment `select from where group by having` de SQL, vers une algèbre relationnelle imbriquée étendue (NRA^{e}) permettant la manipulation des environnement au niveau du langage. Cette traduction supporte tous les aspects de SQL_{Alg}, notamment la gestion des environnements et les valeurs nulles. Elle est mécanisée en Coq. Cette mécanisation a permis de prouver la correction de la traduction en terme de préservation de sémantique. Par ailleurs, la mécanisation a été intégrée dans un cadre plus large, celui de DBCert, un compilateur certifié qui compile des requêtes SQL du fragment cité précédemment,vers du code impératif.

Ce travail a permis de démontrer que la sémantique de SQL_{Alg} est capturable par NRA^{e}. Il a jeté le pont entre SqlCert et Q*Cert, qui ont été conçus pour des objectifs et des méthodologies différents. En effet, SQL_{Coq} fait partie d'un projet plus large de travaux de formalisation de langages centrés données, son objectif est de formaliser de manière très rigoureuse la sémantique de SQL. Quant à Q*Cert, l'objectif est plutôt l'optimisation par traductions successives avec l'utilisation de combinateurs. La recherche d'efficacité calculatoire a été prise en compte dans sa construction.

Les points clés de la traduction sont la préservation de la généricité de SQL_{Alg} dans la traduction pour rester orthogonal aux différentes implémentations et l'idée de *découper* les environnements de SQL_{Alg} en deux parties. L'une abstraite dont dépend directement la traduction et l'autre concrète qui est encodée dans un environnement NRA^{e}. Ceci permet de refléter de manière précise dans la traduction comment les environnements d'évaluation changent durant l'évaluation d'une requête SQL. Les traductions des formules et des expressions sont également certifiées. Elles sont indépendantes et peuvent être utilisées dans un autre cadre.

Un travail de recherche à réaliser à court terme concerne l'optimisation des traductions. Actuellement, des optimisations minimes sont effectuées directement au niveau syntaxique. De plus certains opérateurs abstraits ont été définis de sorte que les preuves Coq puissent être réalisées. Des opérateurs équivalents à ces opérateurs mais plus optimaux en terme de temps d'évaluation devraient être définis. D'autres optimisations sont également possibles [3].

Ce travail de thèse et plus largement le projet DBCert posent une première pierre de ce qui peut devenir une plate-forme d'interopérabilité de langages centrés données. En effet, DBCert offre un cadre formel pour la définition et la vérification de systèmes et de langages centrés données. La cohabitation de différents langages dans ce cadre permet d'étudier finement les frontières et les différences entre les différents paradigmes et langages. Une première piste de recherche oriente vers la formalisation et l'intégration dans DBCert d'autres langages centrés données, spécifiquement les langages NoSQL qui ont une sémantique qui n'est pas complètement formelle et qui évolue encore.

3. Par exemple, pour éliminer certaines parties superflues du code engendrées uniquement par l'encodage.

Une autre perspective prometteuse se situe au niveau du *backend*. En effet, actuellement le code impératif produit est compilé vers du JavaScript. L'utilisation d'autres langages en sortie est une piste de travail.

A long terme, en prenant en compte les aspects ergonomiques et en rendant opaques certaines parties de DBCert, il serait très intéressant de mettre ce genre d'outils formels entre les mains d'un plus large public, non expert en méthodes formelles.

Références bibliographiques

[Abiteboul et Bidoit, 1986] Abiteboul, S. et Bidoit, N. (1986). Non first normal form relations : An algebra allowing data restructuring. *Journal of Computer and System Sciences*, 33(3):361–393.

[Auerbach *et al.*, 2017a] Auerbach, J., Hirzel, M., Mandel, L., Shinnar, A. et Siméon, J. (2017a). Prototyper un compilateur de requêtes avec Coq. *In 28èmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, pages 101–116, Gourette, France.

[Auerbach *et al.*, 2017b] Auerbach, J. S., Hirzel, M., Mandel, L., Shinnar, A. et Siméon, J. (2017b). Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. *In* Salihoglu, S., Zhou, W., Chirkova, R., Yang, J. et Suciu, D., éditeurs : *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1555–1569. ACM.

[Auerbach *et al.*, 2017c] Auerbach, J. S., Hirzel, M., Mandel, L., Shinnar, A. et Siméon, J. (2017c). Prototyping a query compiler using coq (experience report). *Proc. ACM Program. Lang.*, 1(ICFP).

[Auerbach *et al.*, 2017d] Auerbach, J. S., Hirzel, M., Mandel, L., Shinnar, A. et Siméon, J. (2017d). Q*cert : A platform for implementing and verifying query compilers. *In* Salihoglu, S., Zhou, W., Chirkova, R., Yang, J. et Suciu, D., éditeurs : *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1703–1706. ACM.

[Bancilhon *et al.*, 1982] Bancilhon, F., Richard, P. et Scholl, M. (1982). On line processing of compacted relations. *In VLDB*, pages 263–269.

[Benzaken et Contejean, 2019] Benzaken, V. et Contejean, É. (2019). A coq mechanised formal semantics for realistic SQL queries : formally reconciling SQL and bag relational algebra. *In Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 249–261.

[Benzaken *et al.*, 2014] Benzaken, V., Contejean, E. et Dumbrava, S. (2014). A coq formalization of the relational data model. *In* Shao, Z., éditeur : *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of*

the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, volume 8410 de *Lecture Notes in Computer Science*, pages 189–208. Springer.

[Benzaken *et al.*, 2017] Benzaken, V., Contejean, É. et Dumbrava, S. (2017). Certifying standard and stratified datalog inference engines in ssreflect. *In International Conference on Interactive Theorem Proving*, pages 171–188. Springer.

[Benzaken *et al.*, 2018] Benzaken, V., Contejean, E., Keller, C. et Martins, E. (2018). A coq formalisation of sql's execution engines. *In* Avigad, J. et Mahboubi, A., éditeurs : *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 de *Lecture Notes in Computer Science*, pages 88–107. Springer.

[Bertot et Castéran, 2010] Bertot, Y. et Castéran, P. (2010). *Interactive Theorem Proving and Program Development : Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st édition.

[Bodin *et al.*, 2014] Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A. et Smith, G. (2014). A Trusted Mechanised JavaScript Specification. *In POPL 2014 - 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, United States.

[Cao et Badia, 2007] Cao, B. et Badia, A. (2007). Sql query optimization through nested relational algebra. *ACM Trans. Database Syst.*, 32(3):18–es.

[Chamberlin, 2012] Chamberlin, D. (2012). Early history of sql. *Annals of the History of Computing, IEEE*, 34:78–82.

[Chamberlin *et al.*, 1976] Chamberlin, D. D., Astrahan, M. M., Eswaran, K. P., Griffiths, P. P., Lorie, R. A., Mehl, J. W., Reisner, P. et Wade, B. W. (1976). Sequel 2 : a unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6):560–575.

[Chamberlin et Boyce, 1974] Chamberlin, D. D. et Boyce, R. F. (1974). SEQUEL : A structured english query language. *In Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes*, pages 249–264.

[Cheney et Urban, 2011] Cheney, J. et Urban, C. (2011). Mechanizing the metatheory of mini-xquery. *In* Jouannaud, J.-P. et Shao, Z., éditeurs : *Certified Programs and Proofs*, pages 280–295, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Chu *et al.*, 2017] Chu, S., Weitz, K., Cheung, A. et Suciu, D. (2017). Hottsql : Proving query rewrites with univalent sql semantics. *In PLDI 2017*, pages 510–524, New York, NY, USA. ACM.

[Cluet et Moerkotte, 1993] Cluet, S. et Moerkotte, G. (1993). Nested queries in object bases. *In Database Programming Languages (DBPL-4)*, pages 226–242. Springer.

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks.

Commun. ACM, 13(6):377–387.

[Codd, 1971] Codd, E. F. (1971). Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909.

[Colby, 1990] Colby, L. S. (1990). A recursive algebra for nested relations. *Information Systems*, 15(5):567–582.

[Dayal,] Dayal, U. Of nests and trees : A unified approach to process queries that contain nested subqueries, aggregates, and quantifiers. *In Proceedings of*, volume 1.

[Deux et al., 1990] Deux et al., O. (1990). The story of o2. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):91–108.

[Díaz *et al.*, 2020] Díaz, T., Olmedo, F. et Tanter, E. (2020). A mechanized formalization of graphql. *In Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 201–214, New York, NY, USA. Association for Computing Machinery.

[Ganski et Wong, 1987] Ganski, R. A. et Wong, H. K. (1987). Optimization of nested sql queries revisited. *ACM SIGMOD Record*, 16(3):23–33.

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

[Genevès et Vion-Dury, 2004] Genevès, P. et Vion-Dury, J.-Y. (2004). XPath Formal Semantics and Beyond : a Coq based approach. *In Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logic : TPHOLs 2004*, pages 181–198, Utah, United States. University of Utah.

[Gonthier *et al.*, 2013] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E. et Théry, L. (2013). A Machine-Checked Proof of the Odd Order Theorem. *In* Blazy, S., Paulin, C. et Pichardie, D., éditeurs : *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 de *LNCS*, pages 163–179, Rennes, France. Springer.

[Gonzalia, 2003] Gonzalia, C. (2003). Towards a formalisation of relational database theory in constructive type theory. *In RelMiCS*, pages 137–148.

[Gonzalia, 2006] Gonzalia, C. (2006). *Relations in Dependent Type Theory*. Thèse de doctorat, Chalmers Göteborg University.

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

[Janssen, 1998] Janssen, T. M. V. (1998). Algebraic translations, correctness and algebraic compiler construction. *In Proceedings of the First International AMAST Workshop on Algebraic Methods in Language Processing*, AMiLP '95, page 25–56, NLD. Elsevier Science Publishers B. V.

[Kemper et Moerkotte, 1990] Kemper, A. et Moerkotte, G. (1990). Advanced query pro-

cessing in object bases using access support relations. *In Proceedings of the Sixteenth International Conference on Very Large Databases*, page 290–301, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Kim, 1982] Kim, W. (1982). On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469.

[Leroy, 2019] Leroy, X. (2019). *Le logiciel, entre l'esprit et la matière*, volume 284 de *Leçons inaugurales du Collège de France*. OpenEdition Books.

[Leroy *et al.*, 2016] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M. et Ferdinand, C. (2016). CompCert – a formally verified optimizing compiler. *In ERTS 2016 : Embedded Real Time Software and Systems*. SEE.

[Makinouchi, 1977] Makinouchi, A. (1977). A consideration on normal form of not-necessarily-normalized relation in the relational data model. *In Proceedings of the Third International Conference on Very Large Data Bases, October 6-8, 1977, Tokyo, Japan*, pages 447–453. IEEE Computer Society.

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

[Melton, 2016] Melton, J. (2016). Iso/iec 9075 :2016 information technology - database languages - sql. *ISO/IEC*.

[Morris, 1973] Morris, F. L. (1973). Advice on structuring compilers and proving them correct. *In Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 144–152.

[Mosses, 1980] Mosses, P. (1980). A constructive approach to compiler correctness. *DAIMI Report Series*, 9(118).

[Polak, 1981] Polak, W. (1981). *Compiler Specification and Verification*. Computer Science Department. Stanford University.

[Scholl *et al.*, 1989] Scholl, M., Abiteboul, S., Bancilhon, F., Bidoit, N., Gamerman, S., Plateau, D., Richard, P. et Verroust, A. (1989). Verso : A database machine based on nested relations. *In Nested relations and complex objects in databases*, pages 27–49. Springer.

[Seshadri *et al.*, 1996] Seshadri, P., Pirahesh, H. et Leung, T. C. (1996). Complex query decorrelation. *In Proceedings of the Twelfth International Conference on Data Engineering*, pages 450–458. IEEE.

[Thatcher *et al.*, 1980] Thatcher, J. W., Wagner, E. G. et Wright, J. B. (1980). More on advice on structuring compilers and proving them correct. *In* Jones, N. D., éditeur : *Semantics-Directed Compiler Generation*, pages 165–188, Berlin, Heidelberg. Springer Berlin Heidelberg.

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

[Van Gucht et Fischer, 1988] Van Gucht, D. et Fischer, P. C. (1988). Multilevel nested relational structures. *Journal of Computer and System Sciences*, 36(1):77–105.

Annexes

Annexe

A

Matériel supplémentaire

Un fichier README.md indique comment compiler et utiliser DBCert.

Le code est organisé en 6 répertoires :

— Le répertoire datacert contient la mécanisation de SQL_{Alg} présentée dans [Benzaken et Contejean, 2019]. Cette mécanisation est étendue aux nombres flottants.

— Le répertoire qcert contient le projet Q*Cert, notamment la mécanisation de NRA^e [Auerbach *et al.*, 2017b] et la mécanisation du langage Imp présenté dans le cadre du projet DBCert.

— Le répertoire jsql contient la traduction certifiée présentée dans cette thèse.

— Le répertoire tests contient plusieurs exemples et les expérimentations présentées dans V.1.4.

Description	**Code**	**Thèse**
Traduction de SQL_{Alg} vers NRA^e :	jsql/	IV
- Spécification du modèle de données	jsql/data/TupleToData.v	IV.2
- Spécification de l'encodage de l'instance	jsql/instance/InstanceToNRAEnv.v	IV.3
- Traduction des environnements	jsql/env/EnvToNRAEnv.v	IV.4
- Traduction des expressions simples	jsql/term/FTermToNRAEnv.v	IV.5.1.1
- Traduction des expressions complexes	jsql/term/ATermToNRAEnv.v	IV.5.1.2
- Translation of formulas	jsql/formula/FormulaToNRAEnv.v	IV.5.2
- Traduction des requêtes	query/QueryToNRAEnv.v	IV.5.3
- Théorème de correction	jsql/query/QueryToNRAEnv.v, l.3115	4.2
- Théorème de décodage	jsql/decode/Decode.v, l.73	4.6
Réalisation :		IV.6
- Réalisation du modèle de données	jsql/data/TnullTD.v	IV.6.1
- Réalisation de l'encodage de la logique trivaluée	jsql/formula/TnullBD.v	IV.6.2
- Réalisation de la traduction des fonctions	jsql/formula/TnullFTN.v	IV.6.3
- Réalisation de la traduction des agrégats	jsql/formula/TnullATN.v	IV.6.4
- Réalisation des opérateurs logiques abstraits	jsql/formula/TnullFN.v	IV.6.5
- Réalisation de la spécification globale	query/TnullQN.v	-
Benchmarks sémantiques :		V.1.4.2
- Exemples (valeurs nulles)	tests/null	-
- Examples (requêtes corrélées)	tests/nested	-
Benchmarks de performances :		V.1.4.4
- Bases de données avec 58,800 entrées :	tests/simple/orgX.sql, tests/simple/db1big.json	-

Table A.1. *Emplacement dans le code*

La table A.1 fait le lien entre la thèse et le code.

Annexe

B

Syntaxe d'une requête SELECT

Ci-dessous la syntaxe détaillée d'une requête `select` dans postgreSQL.

Syntaxe d'une requête `select`

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
[ * | expression [ [ AS ] output_name ] [, ...] ]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW window_name AS ( window_definition ) [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
    LAST } ] [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start [ ROW | ROWS ] ]
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
[ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [,
    ...] ] [ NOWAIT | SKIP LOCKED ] [...] ]
```

Syntaxe de from_item

```
[ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ] |
[ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed )
    ] ] |
[ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ] |
with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ] |
[ LATERAL ] function_name ( [ argument [, ...] ] ) |
[ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ] |
[ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias (
    column_definition [, ...] ) |
[ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition
     [, ...] ) |
[ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS (
    column_definition [, ...] ) ] [, ...] ) |
[ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ] |
from_item [ NATURAL ] join_type from_item [ ON join_condition | USING (
    join_column [, ...] ) ]
```

Syntaxe de grouping_element

```
( ) |
expression |
( expression [, ...] ) |
ROLLUP ( { expression | ( expression [, ...] ) } [, ...] ) |
CUBE ( { expression | ( expression [, ...] ) } [, ...] ) |
GROUPING SETS ( grouping_element [, ...] )
```

Syntaxe de with_query

```
with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert
     | update | delete )
```

Annexe

C

Sémantique opérationnelle de NRA^e

$$\frac{}{\gamma \vdash d_0 \,@\, d \Downarrow_a d_0}\ Constant \qquad \frac{}{\gamma \vdash \mathtt{In} \,@\, d \Downarrow_a d}\ ID \qquad \frac{\gamma \vdash q_1 \,@\, d_0 \Downarrow_a d_1 \quad \gamma \vdash q_2 \,@\, d_1 \Downarrow_a d_2}{\gamma \vdash q_2 \circ q_1 \,@\, d_0 \Downarrow_a d_2}\ Comp$$

$$\frac{\gamma \vdash q \,@\, d \Downarrow_a d_0 \quad \boxplus\, d_0 = d_1}{\gamma \vdash \boxplus q \,@\, d \Downarrow_a d_1}\ Unary \qquad \frac{\gamma \vdash q_1 \,@\, d \Downarrow_a d_1 \quad \gamma \vdash q_2 \,@\, d \Downarrow_a d_2 \quad \gamma \vdash d_1 \boxtimes d_2 = d_3}{\gamma \vdash q_1 \boxtimes q_2 \,@\, d \Downarrow_a d_3}\ Binary$$

$$\frac{\gamma \vdash q_1 \,@\, d \Downarrow_a \emptyset}{\gamma \vdash \chi_{\langle q_2 \rangle}(q_1) \,@\, d \Downarrow_a \emptyset}\ Map\,\emptyset \qquad \frac{\gamma \vdash q_1 \,@\, d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash q_2 \,@\, d_1 \Downarrow_a d_2 \quad \gamma \vdash \chi_{\langle q_2 \rangle}(s_1) \,@\, d \Downarrow_a s_2}{\gamma \vdash \chi_{\langle q_2 \rangle}(q_1) \,@\, d \Downarrow_a \{d_2\} \cup s_2}\ Map$$

$$\frac{\gamma \vdash q_1 \,@\, d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash q_2 \,@\, d_1 \Downarrow_a \mathtt{true} \quad \gamma \vdash \sigma_{\langle q_2 \rangle}(s_1) \,@\, d \Downarrow_a s_2}{\gamma \vdash \sigma_{\langle q_2 \rangle}(q_1) \,@\, d \Downarrow_a \{d_1\} \cup s_2}\ Sel_{\mathrm{T}}$$

$$\frac{\gamma \vdash q_1 \,@\, d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash q_2 \,@\, d_1 \Downarrow_a \mathtt{false} \quad \gamma \vdash \sigma_{\langle q_2 \rangle}(s_1) \,@\, d \Downarrow_a s_2}{\gamma \vdash \sigma_{\langle q_2 \rangle}(q_1) \,@\, d \Downarrow_a s_2}\ Sel_{\mathrm{F}} \qquad \frac{\gamma \vdash \sigma_{\langle q_2 \rangle}(q_1) \,@\, d \Downarrow_a \emptyset}{\gamma \vdash q_1 \,@\, d \Downarrow_a \emptyset}\ Sel_{\emptyset}$$

$$\frac{\gamma \vdash q_1 \,@\, d \Downarrow_a \emptyset}{\gamma \vdash q_1 \times q_2 \,@\, d \Downarrow_a \emptyset}\ Prod^l_{\emptyset} \qquad \frac{\gamma \vdash q_2 \,@\, d \Downarrow_a \emptyset}{\gamma \vdash q_1 \times q_2 \,@\, d \Downarrow_a \emptyset}\ Prod^r_{\emptyset}$$

$$\frac{\gamma \vdash q_1 \,@\, d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash q_2 \,@\, d \Downarrow_a \{d_2\} \cup s_2 \quad \gamma \vdash \{d_1\} \times s_2 \,@\, d \Downarrow_a s_3 \quad \gamma \vdash s_1 \times (\{d_2\} \cup s_2) \,@\, d \Downarrow_a s_4}{\gamma \vdash q_1 \times q_2 \,@\, d \Downarrow_a \{d_1 \odot d_2\} \cup s_3 \cup s_4}\ Prod$$

$$\frac{}{\gamma \vdash \mathtt{Env} \,@\, d \Downarrow_a \gamma}\ Env \qquad \frac{\gamma_1 \vdash q_1 \,@\, d_1 \Downarrow_a \gamma_2 \quad \gamma_2 \vdash q_2 \,@\, d_1 \Downarrow_a d_2}{\gamma_1 \vdash q_2 \circ^e q_1 \,@\, d_1 \Downarrow_a d_2}\ \mathrm{Comp}^e$$

$$\frac{\gamma \vdash q_1 \,@\, d \Downarrow_a d_1}{\gamma \vdash q_1 \,\oplus\, q_2 \,@\, \mathtt{left}\ d \Downarrow_a d_1}\ \mathrm{Either}_{\mathtt{left}} \qquad \frac{\gamma \vdash q_2 \,@\, d \Downarrow_a d_2}{\gamma \vdash q_1 \,\oplus\, q_2 \,@\, \mathtt{right}\ d \Downarrow_a d_2}\ \mathrm{Either}_{\mathtt{right}}$$

Annexe

D

Spécification paramétrant SQL_{Alg}

```
Record Rcd : Type :=
  mk_R
    {
      type : Type; attribute : Type; value : Type;
      predicate : Type; symbol : Type; aggregate : Type;
      (** Abstract tuples *)
      tuple : Type;
      (** particular values *)
      default_value : type → value;
      (** comparisons *)
      OAtt : Oset.Rcd attribute;
      A : Fset.Rcd OAtt;
      OVal : Oset.Rcd value;
      FVal : Fset.Rcd OVal;
      OPred : Oset.Rcd predicate;
      OSymb : Oset.Rcd symbol;
      OAgg : Oset.Rcd aggregate;
      OTuple : Oeset.Rcd tuple;
      CTuple : Fecol.Rcd OTuple;
      (** Typing attributes and values. *)
      type_of_attribute : attribute → type;
      type_of_value : value → type;
      (** labels and field extraction *)
      labels : tuple → Fset.set A;
      dot : tuple → attribute → value;
      mk_tuple : Fset.set A → (attribute → value) → tuple;
      (** Interpretation *)
      B : Bool.Rcd;
      interp_predicate : predicate → list value → Bool.b B;
      interp_symbol : symbol → list value → value;
      interp_aggregate : aggregate → list value → value;
      (** Properties *)
      labels_mk_tuple : ∀s f, labels (mk_tuple s f) =S= s;
      dot_mk_tuple :  ∀a s f,
          dot (mk_tuple s f) a = if a inS? s then f a else default_value (
     type_of_attribute a);
      tuple_eq :  ∀t1 t2,
          (Oeset.compare OTuple t1 t2 = Eq) ↔
          (labels t1 =S= labels t2 ∧ ∀a, a inS (labels t1) → dot t1 a = dot t2 a)
    }.
```

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Traduction mécanisée et certifiée en Coq d'une algèbre relationnelle étendue pour SQL vers une algèbre imbriquée

Mots clés : Formalisation de langages centrés données ; Sémantique formelle de SQL ; Compilation certifiée en Coq.

Résumé : En 1974, Boyce et Chamberlin ont créé le langage SQL en se basant sur l'algèbre relationnelle proposée par Codd en 1970, mais à mesure d'extensions, la sémantique formelle de SQL s'est éloignée de celle de l'algèbre relationnelle. Le petit fragment `select from where` de SQL peut correspondre à une algèbre relationnelle avec une sémantique multiensemble en restreignant les expressions et les formules à celles exprimables en algèbre relationnelle. Pour capturer la sémantique du fragment beaucoup plus réaliste `select from where group by having` en prenant en compte toutes les expressions y compris celles avec agrégats, toutes les formes de formules, les valeurs nulles et, encore plus subtil, les environnements très particuliers de SQL, Benzaken et Contejean ont proposé l'algèbre SQL_{Alg} qui est une extension de l'algèbre relationnelle avec un nouvel opérateur pour la partie `group by having` conçu spécifiquement pour prendre en compte tous les aspects de SQL cités précédemment. Ce même fragment de SQL, avec toute ses subtilités, est-il capturable par l'algèbre relationnelle imbriquée ? Cette thèse prouve formellement que oui. En effet, nous proposons une traduction, certifiée en Coq, de SQL_{Alg} vers NRA^e, qui est une formalisation en Coq de l'algèbre relationnelle imbriquée. La traduction prend en compte les expressions simples et complexes, les formules SQL et reflète parfaitement comment les environnements sont construits et manipulés, spécialement pour les agrégats et les requêtes corrélées. Ce travail s'inscrit dans un cadre plus global, celui du projet DBCert : une chaîne de compilation certifiée en Coq de SQL vers JavaScript.

Title : A Coq certified translation from an extension of relational algebra for SQL to a nested algebra.

Keywords : Formalisation of data centric languages ; SQL's formal semantics ; Certified compilation in Coq.

Abstract : In 1974, Boyce and Chamberlin created SQL using the concepts of the relational algebra proposed by Codd in 1970, but as it evolved, its formal semantics became more and more complex. The small fragment `select from where` of SQL can be mapped to a relational algebra, with bag's semantics and by restricting expressions and formulae to those which can be expressed in relational algebra. To capture the semantics of the much more realistic fragment `select from where group by having`, taking into account all the expressions including those with aggregates, all forms of formulas, null values and, even more subtle, the specific SQL's environments, Benzaken and Contejean propose SQL_{Alg} which is an extension of relational algebra with a new operator for the `group by having` part designed specifically to take into account all the aspects of SQL cited above. Can this same fragment of SQL, with all its subtleties, be captured by nested relational algebra ? The present work formally proves that the answer to this question is yes. Indeed, we have built a certified translation, formalized in Coq, from sqlalg to NRA^e, which is a formalization in Coq of the nested relational algebra. The translation supports simple and complex expressions, SQL's formulae and perfectly reflects how environments are built and manipulated, especially for aggregates and for correlated queries. This work is part of a more global framework : the DBCert project which is a compilation chain certified in Coq from SQL to JavaScript.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France