Bibliothèque certifiée en Coq pour la provenance des données

Rébecca Zucchini

université PARIS-SACLAY

Bibliothèque certifiée en Coq pour la provenance des données

A Coq certified library for data provenance

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : ISN. Référent : Falculté des Sciences d'Orsay

Thèse préparée au Laboratoire Méthodes Formelles (Université Paris-Saclay, CNRS, ENS Paris-Saclay), sous la direction de Véronique BENZAKEN, professeure des universités, la co-direction de Sarah COHEN-BOULAKIA, professeure des universités et la co-direction de Chantal KELLER, maîtresse de conférences

Thèse soutenue à Paris-Saclay, le 4 juillet 2023, par

Rébecca Zucchini

Composition du jury

Membres du jury avec voix délibérative

Nicolas THIERY Professeur des universités, LISN	Président
Yves BERTOT Directeur de recherche, Inria Sophia Antipolis	Rapporteur & Examinateur
Pierre SENELLART Professeur des universités, École normale supérieure	Rapporteur & Examinateur
Catherine DUBOIS Professeure des universités,ENSIIE	Examinatrice
Assia MAHBOUBI Directrice de recherche, Inria Nantes/ LS2N	Examinatrice

THESE DE DOCTORAT

NNT : 2023UPASG040

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Bibliothèque certifiée en Coq pour la provenance des données
Mots clés : Bibliothèque certifiée, Provenance, Assistant de preuves, Coq, Base de données

Résumé : La présente thèse se situe à l'intersection des méthodes formelles et des bases de données, et s'intéresse à la formalisation de la provenance des données à l'aide de l'assistant de preuve Coq. L'étude de la provenance des données, qui permet de retracer leur origine et leur historique, est essentielle pour assurer la qualité des données, éviter les interprétations erronées et favoriser la transparence dans le traitement des données. La thèse propose ainsi la formalisation de deux types de provenance des données couramment utilisés, la How-provenance et la Where-provenance. Cette formalisation a permis de comparer leur sémantique, mettant en évidence leurs différences et leurs complémentarités. En outre, elle a conduit à la proposition d'une structure algébrique et d'une sémantique unificatrice pour ces deux types de provenance.

Title : A Coq certified library for data provenance
Keywords : Certified library, Proof assistant, Databases, Coq, Provenance

Abstract : This thesis is about the formalization of data provenance using the Coq proof assistant, at the intersection of formal methods and database communities. It explores the importance of data provenance, which tracks the origin and history of data, in addressing issues such as poor data quality, incorrect interpretations, and lack of transparency in data processing. The thesis proposes formalizations of two commonly used types of data provenance, How-provenance and Where-provenance. Formalizing both types of provenance allowed us to compare their semantics, highlighting their differences and complementarities. Additionally, the formalization of these two types of provenance led to the proposal of an algebraic structure that provides a unifying semantics.

Table des matières

Remerciements

Je tiens à exprimer ma profonde gratitude envers toutes les personnes qui ont participé à l'élaboration de ce manuscrit de thèse. Leur soutien et leur engagement ont été essentiels à la réalisation de ce travail.

Mes remerciements vont tout d'abord aux membres de mon jury, Yves Bertot, Pierre Senellart, Catherine Dubois, Assia Mahboubi et Nicolas Thiery, pour avoir accepté de faire partie de mon jury de thèse. Leurs commentaires constructifs et leurs suggestions pertinentes ont grandement contribué à l'amélioration de ce manuscrit.

Je tiens à remercier chaleureusement ma directrice de thèse, Véronique Benzaken, pour sa confiance en mes capacités et pour m'avoir offert l'opportunité d'effectuer ma thèse sous sa direction. Je souhaite également exprimer ma gratitude envers mon encadrante de thèse, Sarah Cohen-Boulakia, pour son expertise dans le domaine des bases de données biologiques, ainsi que pour son soutien infaillible tout au long de ces quatre années de recherche. Je la remercie également pour sa bonne humeur contagieuse et d'avoir osé s'aventurer dans le monde mystérieux des assistants de preuves. Enfin, je tiens à exprimer ma reconnaissance toute particulière envers ma troisième encadrante de thèse, Chantal Keller, pour sa patience lors de mes explications détaillées sur des points problématiques, son soutien réunion après réunion et ses encouragements lors des moments de doute. Je tiens à la remercier sincèrement pour ses nombreux conseils inestimables (j'utiliserai encore longtemps de nombreux exemples dans mes travaux). Avoir travaillé sous sa supervision a été un grand plaisir et une immense chance.

Je souhaite également exprimer mon appréciation envers mes collègues du LMF, en particulier l'ancienne équipe VALS, pour les pauses café très agréables et pour avoir supporté les excentricités de certaines doctorantes. Je remercie mes amis et collègues, Alexandrina, Yaëlle et Antoine, pour leur soutien moral, nos moments de décompression et nos ateliers collage de post-it. Cette thèse n'aurait pas eu la même saveur sans vous. Je tiens également à remercier mes amis qui ont fait l'ENS avec moi (et parfois aussi la prépa) pour leur soutien constant.

Enfin, je souhaite exprimer ma gratitude envers ma famille et mes proches pour leur support constant et leur amour inconditionnel. Leur foi en mes capacités et leurs encouragements m'ont donné la force nécessaire pour surmonter les obstacles rencontrés tout au long de ce parcours académique.

En conclusion, je suis consciente que cette thèse n'aurait pas été possible sans le soutien et les contributions de toutes les personnes mentionnées ci-dessus. Leur apport a été inestimable et je leur en suis profondément reconnaissante.

1 - Introduction

Cette thèse se trouve à l'intersection entre deux domaines de l'informatique : le monde des méthodes formelles et le monde des bases de données. A priori, ces deux communautés peuvent sembler éloignées dans leurs enjeux et problématiques. Cependant, ces dernières années, il y a eu des résultats notables du côté de la communauté des méthodes formelles proposant des modèles vérifiés de systèmes issus d'autres domaines de l'informatique comme le projet Compcert [1]. Celui-ci propose une vérification formelle d'un compilateur C. Il est donc naturel que la communauté des méthodes formelles se soit attaquée à la formalisation des structures propres aux bases de données. Cette thèse se place à la périphérie de ces travaux et propose d'étudier la formalisation de la provenance des données à l'aide de l'assistant de preuve Coq.

La provenance est apparue il y a 30 ans pour répondre à un besoin grandissant de connaître l'origine d'une donnée et de tracer son histoire lors du traitement d'une requête. Les données se trouvaient jusque-là dans des structures centralisées, ce qui laissait aux gestionnaires du système le contrôle sur les ensembles des données et permettait d'assurer la transparence des opérations effectuées lors des traitements. Il était possible de comprendre les résultats issus de requêtes et d'effectuer des corrections sur celles-ci. Ces environnements très contrôlés par l'utilisateur assuraient que les données manipulées soient de bonne qualité et garantissaient la fiabilité des résultats obtenus. Toutefois, les traitements de données sont actuellement de plus en plus décentralisés et il est maintenant facile et courant d'utiliser des données issus de traitements extérieurs grâce au web. En conséquence, nous sommes passés à des environnements faiblement contrôlés par l'utilisateur : les actions faites sur des données ne sont plus transparentes et sont souvent restreintes. Ce phénomène a été accentué par l'explosion de la génération de données (brutes, intermédiaires ou finales) cette dernière dizaine d'années. Prenons l'exemple d'utilisation d'une analyse de données faite par des chercheurs et chercheuses en bioinformatique. Il est courant pour ces derniers d'avoir recours à des plateformes en ligne comme la plateforme Galaxy [2] pour comparer des séquences de protéines. Ces analyses peuvent comparer plusieurs téraoctets de séquences provenant de sources hétérogènes. Les calculs effectués par ces traitements ne sont pas accessibles aux utilisateurs ni visualisables par eux. L'analyse est donc similaire à une boîte noire et les options pour modifier les calculs sont limitées et opaques.

La perte de contrôle de l'utilisateur sur les traitements de données et l'absence de transparence des opérations effectuées ont un effet domino négatif sur la qualité des données finales d'une analyse, sur la confiance de l'utilisateur dans l'interprétation des données et sur la compréhension des résultats.

La baisse de la qualité des données produites est en corrélation directe avec le manque de transparence des opérations effectuées. Ne pas pouvoir suivre l'utilisation des données lors d'un traitement diminue considérablement la qualité des données obtenues. En effet, il est impossible de savoir si la donnée de qualité la plus médiocre d'une base de données n'a pas affecté le résultat. La qualité de chaque donnée finale ne pourra pas excéder la qualité de cette donnée. Cela a un impact significatif sur certaines structures de traitement des données, en particulier les entrepôts de données chargés d'analyser d'immenses ensembles de données de qualité diverse provenant de multiples sources.

Une conséquence directe de la faible qualité des résultats d'une analyse est la diminution de la confiance accordée à l'interprétation de ces résultats. Les résultats d'un traitement doivent être de bonne qualité pour ne pas introduire d'erreurs d'interprétation. Prenons les données médicales des patients qui sont utilisées pour choisir quel traitement est le plus efficace ou approprié pour un patient donné. Des données de mauvaise qualité peuvent mal orienter les médecins et dégrader la santé du patient. Les résultats devenant non fiables ne sont plus exploitables.

L'absence de transparence des opérations effectuées rend aussi plus difficile la compréhension des résultats. Lors de la manipulation d'un ensemble de données de plusieurs téraoctets, retrouver quelles données ont influencé un certain résultat devient impossible. Les larges volumes de données empêchent d'analyser la raison pour laquelle une donnée se trouve ou non dans le résultat ou encore identifier des erreurs dans un traitement. L'opacité des outils d'analyses de données décentralisées impose aux utilisateurs de travailler en boîtes noires et empêche une compréhension fine des calculs effectués.

Le contrôle et la transparence de ce qui se passe lors d'un traitement de données sont donc des enjeux cruciaux pour la communauté des bases de données.

Pour pallier ce problème, la communauté a introduit le concept de provenance de données. La provenance d'une donnée d consiste à ajouter les méta-données (appelées annotations) qui nous renseignent sur les données ayant eu un impact sur la production de la donnée d et les transformations effectuées sur ces dernières pour obtenir le résultat. Grâce à cet ajout, la provenance augmente ainsi le contrôle et la transparence des calculs effectués.

La littérature propose de découper en plusieurs sous-types la provenance. Chaque type se concentre sur une caractéristique particulière de la provenance et est souvent défini à partir d'une question à laquelle le type s'efforce de répondre. La Why-provenance se concentre sur les raisons qui font qu'une donnée se trouve dans le résultat d'un traitement, et la Why-not-provenance sur celles pour lesquelles une donnée ne s'y trouve pas. La How-provenance s'intéresse aux opérations effectuées sur les données qui aboutissent à un certain résultat. La Where-provenance donne l'ensemble des valeurs ayant eu un impact sur une valeur particulière d'une donnée finale.

L'ajout d'annotations de provenance s'est imposé comme une méthode efficace pour accroître la transparence des opérations sur les données de manière fine. De nombreuses communautés manipulant des données ont confirmé l'utilité de la provenance en les intégrant à leurs outils. Par exemple, la communauté bioinformatique inscrit dans les bons usages à adopter pour assurer la qualité des données le principe que les données soient annotées par leur provenance. Ces bons usages sont regroupés sous la forme des principes "FAIR" (Findable Accessible Interoperable Reusable) [3, 4].

Cependant, les analyses manipulant des données sensibles nécessitent une attention particulière pour préserver leur qualité mais aussi assurer une interprétation dans laquelle nous pouvons avoir confiance. Les utilisateurs demandent ainsi à avoir de fortes garanties sur les informations de provenance des données finales. Au vu de l'importance de la provenance dans la communauté des bases de données, il est donc crucial de s'assurer que la provenance et ses propriétés sont correctes. Même si des cadres mathématiques ont été proposés pour la provenance, peu d'efforts ont été consacrés pour la formaliser dans un outil logiciel offrant des garanties fortes. Dans cette thèse, nous proposons d'utiliser l'assistant de preuve Coq pour formaliser la provenance et ainsi obtenir de telles garanties.

Dans le **Chapitre 2**, nous présentons l'état des connaissances pour la provenance et les formalisations des éléments des bases de données proposées par la communauté des méthodes formelles. Le **Chapitre 3** constitue un rappel des structures et de la sémantique de l'algèbre relationnelle positive avec une extension aux fonctions d'agrégation. Ce chapitre présente aussi la bibliothèque Coq, Datacert, qui formalise les structures des données, des requêtes et la sémantique de l'algèbre relationnelle. Cette bibliothèque constitue la base sur laquelle je m'appuie pour formaliser deux des types de provenance couramment utilisés, la How-provenance (**Chapitre 4**) et la Where-provenance (**Chapitre 5**). Dans chacun de ces chapitres, nous nous placerons dans l'algèbre relationnelle positive qui sera dans certains cas étendue aux fonctions d'agrégation. Nous proposons aussi de vérifier les propriétés propres à chaque type de provenance telles que l'équivalence entre la sémantique de l'algèbre relationnelle positive et celle des annotations plongées dans le semi-anneau des entiers naturels. Le **Chapitre 4** a par ailleurs fait l'objet d'une publication [5]. La formalisation de ces deux types de provenance permet d'avoir une vision variée de la provenance et dans le **Chapitre 6** une meilleure comparaison de la sémantique de ces deux types de provenance. Ce chapitre propose aussi une structure algébrique et une sémantique unificatrice de la How-provenance et la Where-provenance.

Un effort conséquent a été porté sur la lisibilité des fonctions définies, mais aussi à minimiser la longueur des preuves et à mettre en lumière l'enchaînement des lemmes. Cela a demandé parfois de retravailler certaines structures proposées et de simplifier les lemmes et les preuves. Le tableau qui suit indique le nombre de

lignes de code développées pour chaque chapitre :

Chapitre	Section	Lignes
Chapitre 3	**Sous-section 3.4.1**	757
	Sous-section 3.4.2	2741
Chapitre 4	**Section 4.2**	1087
	Section 4.3	532
	Section 4.4	381
	Section 4.5	1916
	Section 4.6	3573
	Section 4.7	1622
Chapitre 5	**Section 5.2** et **Section 5.3**	3755
	Section 5.4	911
Chapitre 6	**Section 6.3**	2771
	Section 6.4	3952
	Section 6.5	252
Total		24250

Dans les différents chapitres, je présenterai la plupart du temps l'intuition derrière les théorèmes énoncés mais je m'attarderai en particulier sur la preuve du théorème d'adéquation de la **Section 4.6**.

2 - État de l'Art

Un peu d'histoire

Un des premiers travaux proposant un modèle de traçage de la provenance des données est proposé dans [6], bien qu'à cette époque, les auteurs ne mentionnent pas encore le terme "provenance". Ce travail s'inscrit dans le contexte du modèle relationnel avec des données hétérogènes provenant de multiples sources. Les auteurs cherchent, avec leur modèle *polygen*, à aborder deux problèmes : le problème du *Data Source Tagging* - c'est-à-dire obtenir la localisation des sources d'une donnée et *Intermediate Source Tagging* - connaître la localisation des données devant vérifier certaines propriétés pour acquérir une donnée finale particulière. Ce premier essai de traçage des données pose déjà les bases de la définition de la provenance.

A la suite de [6] est apparu quelques années plus tard le concept de *lineage*. Une définition en est proposée dans [7] dans le cadre des vues relationnelles avec fonctions d'agrégation dans les entrepôts de données. Le lineage a pour but d'identifier les données initiales d'une base de données qui contribuent à la production d'une donnée finale particulière.

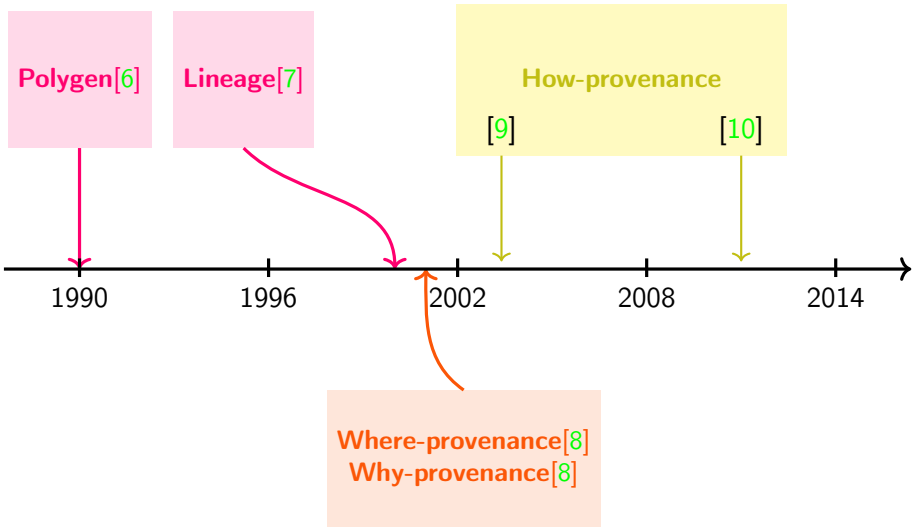
En 2001, le travail [8] présente deux sous-types de provenance, la *why-provenance* et la *where-provenance*, pour des modèles de données semi-structurées.

Le lineage et la why-provenance se distinguent sur le niveau de précision des informations recueillies. Le lineage nous donne les données ayant contribué à la production d'une donnée d, mais ne précise pas si ces données doivent coexister dans la base de données pour que d soit présente dans le résultat ou si certains sous-ensembles de ces données suffisent. En revanche, la why-provenance fait cette différence grâce à la notion de *witness basis* (base de témoins). Un témoin d'une donnée résultante d est une donnée source qui contribue à la présence de d dans le résultat d'une requête. Un élément de la base de témoins d'une donnée d correspond à un ensemble de témoins de d suffisant pour assurer que leur coexistence dans la base de données garantit la présence de d dans le résultat et l'absence de l'un d'entre eux supprime d des résultats. La why-provenance d'une donnée correspond à ces ensembles qui "témoignent" de la présence de cette donnée dans le résultat. Elle répond ainsi à la question "pourquoi une donnée apparaît dans le résultat ?" en citant les groupes de données sources produisant cette donnée.

La where-provenance, quant à elle, suit la propagation d'une valeur d'une donnée lors d'une requête. Dans ce cadre, chaque valeur initiale est marquée/étiquetée et lors de l'application d'une requête, ces annotations sont propagées de la valeur initiale vers les emplacements où elle est utilisée.

Les travaux décrits dans [9] proposent une nouvelle approche qui consiste à plonger les annotations des données dans des structures nommées *semi-anneaux*

commutatifs. Ces travaux se placent dans le cadre de l'algèbre relationnelle positive ensembliste. Ils introduisent également le concept de K-relations, qui associe à chaque n-uplet d'une relation son annotation. La structure de semi-anneau peut être instanciée en fonction des besoins de l'utilisateur pour retenir certaines informations. Selon le semi-anneau choisi, la sémantique de l'algèbre relationnelle positive annotée coïncide avec la sémantique des bases de données incomplètes, probabilistes, calculant la why-provenance ou encore à une sémantique multi-ensembliste des bases de données. Cette approche annotant les n-uplets par des éléments plongés dans des semi-anneaux correspond à la how-provenance qui répond à la question "comment les données ont été transformées lors d'une requête". Dans un travail plus récent [10], la *how-provenance* est étendue aux fonctions d'agrégation comme "min", "max", "count" ou encore "sum".



Chacun des sous-types de provenance citées ci-avant ont été intégrés dans des systèmes de gestion de base de données. Nous pouvons citer par exemple l'implémentation du système de gestion Trio [11] qui supporte le lineage, du module ProvSQL [12] pour le système de gestion de bases de données PostgreSQL supportant la how-provenance pour l'algèbre relationnelle ou encore DBnotes [13] qui peut tracer la where-provenance [14].

Dans cette thèse, nous nous attachons à formaliser dans l'assistant de preuve Coq la how-provenance et la where-provenance, deux types de provenance couramment utilisés. Ces deux types de provenance ont en commun d'être basés sur la propagation d'annotations à grains fins et de se concentrer sur les résultats obtenus après l'application d'une requête. Même si les types de provenance qui ont été cités précédemment ne sont pas forcément restreints à ce cadre, le travail de cette thèse se concentre sur l'algèbre relationnelle positive [15] et dans certains cas, une extension avec des fonctions d'agrégation sera étudiée.

Provenance et annotations

Le traçage d'informations de provenance lors de l'application d'une requête faite à l'aide d'annotations est une approche de haut en bas de la provenance. Des approches de bas en haut ou n'utilisant pas les annotations sont proposées dans la littérature comme dans le travail [16]. Ses auteurs approximent l'ensemble des données ayant contribué au résultat à l'aide d'une fonction de faible inversion - c'est-à-dire des fonctions pour lesquelles on ne possède pas une fonction inverse complète mais seulement une approximation. Par ailleurs, parmi les types de provenance déjà énoncés, la why-provenance est une approche non orientée sur les annotations.

La granularité de la where-provenance et de la how-provenance est très fine : les annotations sont propagées de façon précise entre chaque étape d'une requête, impliquant uniquement les annotations des données ayant un impact direct sur le résultat final de la requête. Cette granularité n'est pas toujours applicable lorsque les outils utilisés ne proposent pas une explication détaillée de leur fonctionnement (boîtes noires). C'est le cas souvent dans le cadre des workflows scientifiques. La provenance pour les workflows contient des informations [17, 18] comme la version de l'outil utilisé ou l'ensemble des données utilisées en entrée mais reste moins précise que la how-provenance ou la where-provenance.

Provenance des données

Les types de provenance abordés dans cette thèse sont tournés vers les données qui sont présentes dans le résultat d'une requête. Un modèle s'intéressant à expliquer pourquoi une donnée n'est pas dans le résultat est proposé dans [19], qui définit la why-not-provenance. Elle propose à partir d'une question du type : "pourquoi n'y-a-t-il pas de données vérifiant le prédicat S présentes dans le résultat ?" et elle détermine les opérateurs qui ont empêché les données de la base vérifiant le prédicat S d'être propagées jusqu'au résultat.

Formalisation avec un outil logiciel des données

La première tentative de formalisation, en Agda [20], de l'algèbre relationnelle est proposée par [21, 22] tandis que la première formalisation en Coq, complète, du modèle relationnel est proposée par [23] où le modèle de données, l'algèbre, les requêtes «tableaux», la procédure de semi-décision «chase» et les contraintes d'intégrité sont formalisés.

La toute première tentative de vérifier, en Coq, un système de gestion de bases de données relationnelles (SGBDR) est présentée dans [24]. Des propositions de sémantiques mécanisées pour SQL sont introduites dans [25, 26]. Une formalisation d'un moteur de requêtes SQL est fournie dans [27].

À notre connaissance, il n'existe pas de travaux sur la formalisation de la provenance des données au moyen d'assistants de preuve.

Une formalisation des K-relations est donnée dans l'article [28] proposant un outil pour décider de l'équivalence de deux requêtes SQL. À cette fin, HottSQL, une sémantique pour un fragment de SQL, est définie. Cette sémantique se base sur la notion de K-relation. Toutefois, la modélisation des K-relations proposée relaxe la contrainte de finitude de celles-ci. Ainsi, cette proposition peut conduire à la manipulation d'instances de K-relations potentiellement infinies. En outre, la sémantique présentée n'est pas exécutable. Notre approche permet d'avoir une sémantique exécutable, ce qui offre la possibilité d'extraire du code, d'exécuter des tests, etc.

Sur le versant de la formalisation de structures algébriques, de nombreuses propositions ont été développées en Coq, la plus élaborée étant Mathematical Components [29]. La formalisation que je propose des semi-anneaux repose sur la bibliothèque Coccinelle [30] pour deux raisons : la compatibilité avec la formalisation de l'algèbre relationnelle sur laquelle repose ce travail, et, de nouveau, la volonté de fournir une sémantique exécutable.

3 - Datacert

3.1 . Algèbre relationnelle positive

Nous présentons l'algèbre relationnelle positive dans un premier lieu. L'une des raisons à cela est que les notions de provenance discutées plus en aval se basent sur cette restriction de l'algèbre relationnelle qui exclut la différence.

Nous rappelons dans cette section la syntaxe et la sémantique de l'algèbre relationnelle positive et posons les notations utilisées dans la suite de la thèse. Nous nous plaçons dans le modèle multi-ensembliste.

Ce modèle décrit les relations (ou tables) et les opérations qui leur sont appliquées. Une relation est représentée graphiquement sous forme d'un tableau. Chaque ligne (excepté la première) renferme une donnée, aussi nommée n-uplet.

	Patient	**Maladies**
t_1	Pat.1	Rhume
t_2	Pat.51	Varicelle
t_3	Pat.59	Otite
t_4	Pat.1	Otite

Table 3.1 – relation r_0

Par exemple, dans la table r_0, t_1 =(Pat.1,Rhume) est un n-uplet. Chaque colonne est identifiée par un attribut dont le nom est donné dans la première ligne. Les attributs de la table r_0 sont "Patient" et "Maladies". Pour chaque colonne, un n-uplet a une valeur. La valeur de t_1 pour l'attribut "Patient" est Pat.1. Le domaine d'un attribut est l'ensemble dans lequel les données peuvent prendre leurs valeurs. Par exemple pour la colonne "Patient", les valeurs commencent toujours par "Pat." suivi d'un nombre. Le schéma d'une relation indique le nom, les attributs et les domaines des attributs de cette relation.

Une relation ou table est donc un multi-ensemble de n-uplets étant associés aux mêmes attributs. Dans la pratique, les ensembles de données manipulés sont finis : nous supposerons que les relations sont constituées d'un nombre fini de n-uplets. Les tables étant prédéfinies dans une base de données seront ici appelées tables sources.

On utilisera les notations suivantes :

— $\mathcal{D}om$– ensemble des domaines
— $\mathcal{V}al_{\mathscr{D}}$– ensemble des valeurs d'un domaine $\mathscr{D}$
— $\mathcal{A}$– ensemble des attributs possibles
— $\mathcal{T}$ – ensemble des n-uplets possibles
— $\mathcal{R}el$– ensemble des relations possibles
— $\mathcal{N}rel_s$– ensemble des noms de ces tables sources

La notation

$$\{e \in E | P(e)\}$$

définit un ensemble dont les éléments sont les éléments e de l'ensemble E qui vérifient une propriété P.
La notation

$$\{\{(e : n_e) \in M | P(e)\}\}$$

définit un multi-ensemble dont les éléments sont les éléments e du multi-ensemble M qui vérifient une propriété P et chaque élément e apparaît avec une multiplicité n_e.
La fonction $\boldsymbol{nb}(M, e)$ indique aussi le nombre d'occurrences (ou la multiplicité) de l'élément e dans le multi-ensemble M.
On nommera pour la suite l'ensemble des attributs associés à un n-uplet, la sorte de ce n-uplet. De même pour une relation et une requête.

3.1.1 . Requêtes de l'algèbre relationnelle positive

Les opérations sur ces relations se nomment les requêtes. Le langage des requêtes de l'algèbre relationnelle positive s'articule en un cas de base et six opérateurs construits inductivement :

Définition 3.1: Syntaxe (Algèbre relationnelle)

$$q := r | \epsilon_{<>} | \rho_r(q) | \pi_A(q) | \sigma_f(q) | q \bowtie q' | q \cup q'$$

L'application de ces requêtes sur un ensemble de relations prédéfinies effectue des opérations sur ces dernières et renvoie le résultat sous forme d'une nouvelle relation. J'emploierai le terme table initiale ou n-uplet initial pour désigner une table ou un n-uplet provenant de la table associée à une sous-requête de la requête principale. De même, les termes "table résultante" ou "finale" (de même pour n-uplet résultant ou final) correspondent à la table (réciproquement le n-uplet de la table) qui résulte de l'application d'une requête.

La requête "relation r"

Je me permets d'appeler ici requête "relation r", une requête de base dont le rôle est de renvoyer la table correspondant au nom de relation r. Cette requête correspond dans le langage SQL à la requête `Select * from r`. Le nom r doit faire partie des noms des tables sources préalablement définies, c'est-à-dire être dans l'ensemble $\mathcal{N}rel_s$. Les requêtes contenant le nom de relation n'apparaissant pas dans $\mathcal{N}rel_s$ sont considérées comme mal formées.

Les tables r_0, r_1 et r_2 sont pour notre exemple les trois tables sources. Les tables r_0 et r_1 représentent les données de patients atteints d'une maladie et

	Patient	Maladies
t_5	Pat.67	Angine
t_6	Pat.1	Otite
t_7	Pat.314	Gastro

Table 3.2 – relation r_1

	Patient	Médic
t_8	Pat.1	Goutte
t_9	Pat.59	AB[1]
t_{10}	Pat.314	Goutte
t_{11}	Pat.67	Doliprane

Table 3.3 – relation r_2

associent à chaque patient anonymisé le type de maladie dont il est atteint. La table r_2 donne pour chaque patient, les médicaments prescrits : des gouttes (Goutte), des anti-biotiques (AB) ou du doliprane.

La table avec le n-uplet vide $\epsilon_{<>}$

t_{12}

Table 3.4 – $\epsilon_{<>}$

L'opérateur $\epsilon_{<>}$ est l'autre opérateur de base qui renvoie la table contenant un n-uplet vide (noté $<>$), c'est-à-dire dont l'ensemble d'attributs est vide. Elle est représentée par la **Table 3.4**.

Le renommage $\rho_r(\cdot)$

	Patient	Organe
t'_1	Pat.1	Rhume
t'_2	Pat.51	Varicelle
t'_3	Pat.59	Otite
t'_4	Pat.1	Otite

Table 3.5 – renommage $\rho_{[Maladie:=Organe]}(r_0)$

L'opérateur de renommage $\rho_r(\cdot)$ modifie les noms des attributs de la sorte d'une table. r est une fonction qui associe les attributs que l'on veut renommer avec les nouveaux attributs choisis. Dans la **Table 3.5**, l'attribut "Maladies" est renommé en "Organe". Le nom de la deuxième colonne change et les attributs associés au n-uplet t'_1 sont maintenant "Patient" et "Maladies". La syntaxe pour la fonction de renommage est la suivante :

$$r ::= \varnothing | a := b, r$$

où a et b sont des attributs. On dira qu'un attribut est en entrée d'une fonction de renommage si cet attribut est celui qui va être renommé. Un attribut est en sortie lorsqu'il remplace l'ancien attribut.

Une fonction de renommage r est bien formée si les deux conditions suivantes sont vérifiées :

— Les attributs en entrée de la fonction de renommage r sont deux à deux distincts : on ne renomme donc pas deux fois la même colonne.

— Les attributs en sortie de la fonction de renommage r sont deux à deux distincts : on ne veut pas avoir deux colonnes avec le même nom en sortie

De plus, une fonction de renommage est applicable sur la table associée à une requête q si les attributs de la sorte de q sont deux à deux distincts après avoir été renommé par r : cela évite que deux colonnes aient le même nom dans la table résultante. En effet, on pourrait avoir une table dont les attributs sont A et B et renommer A en B. La fonction $A := B$ est applicable mais donnerait deux colonnes avec le même nom.

La projection $\pi_A(\cdot)$

	Maladies
t_{13}	Rhume
t_{14}	Varicelle
t_{15}	Otite
t_{16}	Otite

Table 3.6 – projection $\pi_{Maladie}(r_0)$

L'opérateur unaire de projection $\pi_A(\cdot)$ sur l'ensemble des attributs A conserve une partie de chaque n-uplet de la table associée à la sous-requête : seules les valeurs liées aux attributs contenus dans A sont gardées. Dans la **Table 3.6**, l'attribut sur lequel on projette est "Maladies". Le n-uplet initial t_1 devient alors t_{13} et perd donc l'information "Pat. 1". Les attributs de A n'étant pas présents dans la sorte de la sous-requête sont ignorés et n'ont pas d'impact sur la table résultante.

La sélection $\sigma_f(\cdot)$

	Patient	**Maladie**
t_3	Pat.59	Otite
t_4	Pat.1	Otite

Table 3.7 – sélection $\sigma_{Type=Otite}(r_0)$

L'opérateur unaire de la sélection $\sigma_f(\cdot)$ selon une formule logique f évalue f sur chaque n-uplet t de la table associée à la sous-requête : si la formule est satisfaite par t alors le n-uplet t est conservé dans la table résultante. Dans le cas

contraire, il ne l'est pas. La **Table 3.7** ne conserve de la table r_0 que les n-uplets ayant la valeur "Otite" pour l'attribut "Maladies", c'est-à-dire les n-uplets t_3 et t_4.

Les formules logiques prises en argument par l'opérateur de sélection prennent la forme suivante :

$$f ::= \top | f \wedge f' | f \vee f' | \neg f | P_{\bar{t}_1, \bar{t}_2}$$

où P est un prédicat et où $\bar{t}_1$ et $\bar{t}_2$ sont des termes. Dans notre contexte, les prédicats prennent la forme de comparaisons entre termes qui sont soit des constantes soit des attributs.

La jointure naturelle $\cdot \bowtie \cdot$

	Patient	**Maladie**	**Médic**
t_{17}	Pat.1	Rhume	Goutte
t_{18}	Pat.59	Otite	AB
t_{19}	Pat.1	Otite	Goutte

Table 3.8 – jointure naturelle $r_0 \bowtie r_2$

L'opérateur binaire de la jointure naturelle $\cdot \bowtie \cdot$ fusionne les n-uplets des tables associées aux sous-requêtes. La fusion entre deux n-uplets n'est possible que dans le cas où les valeurs des deux n-uplets sont égales sur les éventuels attributs communs entre les deux tables. Le cas échéant, la fusion des deux n-uplets donne un n-uplet contenant les valeurs des deux n-uplets et sa sorte correspond à l'union des sortes des n-uplets initiaux. Dans la **Table 3.8**, le n-uplet t_{17} est la fusion du n-uplet t_1 et du n-uplet t_8 : l'attribut en commun "Patient" est le même pour les deux n-uplets et la valeur pour l'attribut "Maladies" (respectivement "Médic") est la valeur prise par t_1 (respectivement t_8) pour cet attribut. On remarque que le n-uplet t_2 ne peut fusionner avec aucun des n-uplets de la **Table 3.3** et ainsi le patient "Pat.51" n'apparaît pas dans la **Table 3.8**.

L'union $\cdot \cup \cdot$

L'opérateur binaire de l'union ensembliste $\cdot \cup \cdot$ entre deux sous-requêtes a pour fonction de rassembler les n-uplets contenus dans les sous-requêtes dans une seule relation. Cette requête nécessite que les sous-requêtes aient les mêmes sortes associées. Dans le cas contraire, la requête est mal formée. La **Table 3.9** est la table résultante de l'union des **Tables 3.1 et 3.2**. Elles regroupent tous les n-uplets de la **Table 3.1** (t_1 à t_4) et ceux de la **Table 3.2** (t_5 à t_7).

	Patient	Maladies
t_1	Pat.1	Rhume
t_2	Pat.51	Varicelle
t_3	Pat.59	Otite
t_4	Pat.1	Otite
t_5	Pat.67	Angine
t_6	Pat.1	Otite
t_7	Pat.314	Gastro

Table 3.9 – union $r_0 \cup r_1$

3.1.2 . Sémantique et bonne formation

Dans cette sous-section, nous nous appliquons à définir inductivement la sorte, la bonne formation et la sémantique de chaque opérateur des requêtes. Les fonctions définies pour la sorte et la bonne formation jouent un rôle crucial lors de la formalisation en Coq.

Pour cela, nous posons ici deux définitions en préambule : le renommage d'attribut par une fonction et la projection d'un n-uplet sur un sous ensemble de sa sorte.

Préambule

Commençons par la définition du renommage d'attributs par une fonction de renommage. Une fonction de renommage est une fonction partielle de l'ensemble des attributs vers l'ensemble des attributs. Je définis tout d'abord la bonne formation d'une fonction de renommage.

Définition 3.2: Fonction de renommage bien formée

Une fonction de renommage r est bien formée si :
- $r = \varnothing$ ou
- lorsque r est égale à $a_1 := b_1, .., a_n := b_n$,

$$\forall i, j, i \neq j \Rightarrow a_i \neq a_j \wedge b_i \neq b_j$$

Dans la suite, les fonctions de renommage sont toujours bien formées.

Définition 3.3: Renommage d'attribut et d'ensemble d'attributs

Le renommage d'un attribut a par la fonction r bien formée noté $a[r]$ est

défini comme suit :

$$a[r] = \begin{cases} b & \text{si } a := b \in r \\ a & \text{sinon} \end{cases}$$

Le renommage des attributs de l'ensemble A par la fonction r bien formée est noté $A[r]$ et est défini inductivement comme suit :

- $\varnothing[r] = \varnothing$
- $(A \cup B)[r] = (A[r]) \cup (B[r])$ où A, B sont des ensembles d'attributs et r une fonction de renommage.

On note $Ent(r)$ et $Sor(r)$ respectivement les ensembles d'attributs en entrée et en sortie de la fonction de renommage r.

On remarque que si la fonction de renommage contient en entrée des attributs n'étant pas dans l'ensemble renommé, ceux-ci n'ont aucun impact sur l'ensemble résultant. De plus, à ce niveau, on ne prend pas en compte la condition qui veut que les attributs après renommage soient deux à deux distincts. Cette condition apparait au niveau de la bonne formation des requêtes.

Projection

La projection d'un n-uplet sur un ensemble d'attributs est définie comme suit :

Définition 3.4: Projection d'un n-uplet

Un n-uplet t' est le résultat de la projection du n-uplet t sur l'ensemble d'attributs A si :

- $\forall a \in A \cap \mathbf{Attr_t}(t), val_a(t') = val_a(t)$ où val_a donne la valeur prise par le n-uplet pris en argument sur l'attribut a et $\mathbf{Attr_t}(\cdot)$ donne la sorte de t.
- la sorte de t' est égale à $A \cap \mathbf{Attr_t}(t)$.

Pour la suite, on note $t_{|A}$ la projection du n-uplet t sur l'ensemble A.

Sorte des requêtes

Pour définir la bonne formation et la sémantique des requêtes, nous devons tout d'abord définir la sorte d'une requête à travers la fonction $\mathbf{Attr_q}(\cdot)$. La sorte d'une requête doit dans le cas où cette requête est bien formée coïncider avec la sorte de chacun des n-uplets issus de cette requête.

Définition 3.5: Sorte d'une requête

Soit q, q_1 et q_2 des requêtes, f une formule et A un ensemble d'attributs.

- $\mathbf{Attr_q}(r) = \mathbf{Attr_{rel}}(r)$ où $\mathbf{Attr_{rel}}(\cdot)$ associe à une table sa sorte

— $\mathbf{Attr_q}(\epsilon_{<>}) = \{\}$
— $\mathbf{Attr_q}(\rho_r(q)) = (\mathbf{Attr_q}(q))[r]$
— $\mathbf{Attr_q}(\pi_A(q)) = A \cap \mathbf{Attr_q}(q)$
— $\mathbf{Attr_q}(\sigma_f(q)) = \mathbf{Attr_q}(q)$
— $\mathbf{Attr_q}(q_1 \bowtie q_2) = \mathbf{Attr_q}(q_1) \cup \mathbf{Attr_q}(q_2)$
— $\mathbf{Attr_q}(q_1 \cup q_2) = \mathbf{Attr_q}(q_1)$

On remarque que la sorte de la projection sur un ensemble A est l'ensemble A car on restreint les n-uplets aux attributs de A. La jointure naturelle entre deux n-uplets garde toutes les informations des n-uplets initiaux : la sorte est donc l'union des sortes des sous-requêtes. Pour le renommage, nous devons appliquer le renommage aux attributs contenus dans la sorte de la sous-requête.

Bonne formation des requêtes

La fonction $(\!|\cdot|\!)_{bf}$ récapitule les cas abordés précédemment où une requête appliquée sur un ensemble de relations sources $\mathcal{N}rel_s$ particulier est bien formée. On notera, pour la suite, V lorsque l'évaluation renvoie vrai et F lorsqu'elle renvoie faux.

Définition 3.6: Bonne formation d'une requête

On note q, q_1 et q_2 des requêtes, f une formule et A un ensemble d'attributs. La fonction $(\!|\cdot|\!)_{bf}$ est définie inductivement comme suit :

— $(\!|r|\!)_{bf} = \begin{cases} V & \text{si } r \text{ est le nom d'une relation contenue dans } \mathcal{N}rel_s \\ F & \text{sinon} \end{cases}$
— $(\!|\epsilon_{<>}|\!)_{bf} = V$
— $(\!|\rho_r(q)|\!)_{bf} = \begin{cases} (\!|q|\!)_{bf} & \text{si } r \text{ est bien formée et} \\ & \forall a_1, a_2 \in \mathbf{Attr_q}(q)^2, a_1 \neq a_2 \Rightarrow a_1[r] \neq a_2[r] \\ F & \text{sinon} \end{cases}$
— $(\!|\pi_A(q)|\!)_{bf} = (\!|q|\!)_{bf}$
— $(\!|\sigma_f(q)|\!)_{bf} =$ si la formule f est bien formée pour la sorte de q alors $(\!|q|\!)_{bf}$ sinon F
— $(\!|q_1 \bowtie q_2|\!)_{bf} = (\!|q_1|\!)_{bf}$ et $(\!|q_2|\!)_{bf}$
— $(\!|q_1 \cup q_2|\!)_{bf} = \begin{cases} (\!|q_1|\!)_{bf} \text{ et } (\!|q_2|\!)_{bf} & \text{si } \mathbf{Attr_q}(q_1) = \mathbf{Attr_q}(q_2) \\ F & \text{sinon} \end{cases}$

On remarque pour la sélection que la bonne formation dépend aussi de la bonne formation de la formule logique pour la sorte de la sous-requête. L'évaluation des prédicats dépend des n-uplets de la table associée à la sous requête : les attributs apparaissant dans les prédicats doivent être contenus dans la sorte de la sous-requête.

Sémantique de l'algèbre relationnelle

Nous définissons dans cette sous-partie la sémantique multi-ensembliste de l'algèbre relationnelle dont l'intuition a été détaillée précédemment. Pour chaque requête, on donne le multi-ensemble (donc la table) renvoyé par l'application de la requête sur l'ensemble de relations prédéfinies $\mathcal{N}rel_s$. La sémantique d'une requête q est notée $[\![q]\!]_{rel}$.

Définition 3.7: Sémantique (Algèbre relationnelle)

On suppose les requêtes bien formées. Soit q, q_1 et q_2 des requêtes, f une formule et A un ensemble d'attributs.

- $[\![r]\!]_{rel} = rel(r)$ où rel est la fonction qui associe la table source correspondant au nom de la relation r
- $[\![\epsilon_{<>}]\!]_{rel} = \{\{<>: 1\}\}$ où $<>$ désigne le n-uplet vide
- $[\![\rho_r(q)]\!]_{rel} = \{\{(t : n) | \exists (t' : n) \in [\![q]\!]_{rel}, \forall a \in \mathbf{Attr_t}(t'), t|_{a[r]} = t'|_a \wedge \mathbf{Attr_t}(t) = \mathbf{Attr_t}(t')[r]\}\}$
- $[\![\pi_A(q)]\!]_{rel} = \{\{(t_{|A} : \sum_{t' \text{ tel que } t'_{|A}=t} \boldsymbol{nb}([\![q]\!]_{rel}, t')) | \exists (t : \_) \in [\![q]\!]_{rel}\}\}$
  où $t'_{|A}$ est la projection sur A du n-uplet t'
- $[\![\sigma_f(q)]\!]_{rel} = \{\{(t : n) | \exists (t : n) \in [\![q]\!]_{rel}, [f]_f(t) = V\}\}$
  où $[\cdot]_f$ est la fonction d'évaluation de la formule f sur un n-uplet
- $[\![q_1 \bowtie q_2]\!]_{rel} = \{\{(t : n_1 \times n_2) | \exists (t_1 : n_1) \in [\![q_1]\!]_{rel}, \exists (t_2 : n_2) \in [\![q_2]\!]_{rel}\},$
  $t_1 = t_{|\mathbf{Attr_q}(t_1)} \wedge t_2 = t_{|\mathbf{Attr_q}(t_2)} \wedge \mathbf{Attr_t}(t) = \mathbf{Attr_q}(t_1) \cup \mathbf{Attr_q}(t_2)\}$
  où $\mathbf{Attr_t}(t)$ est la fonction qui associe à un n-uplet sa sorte et $t_{|A}$ est la projection de t sur A
- $[\![q_1 \cup q_2]\!]_{rel} = [\![q_1]\!]_{rel}(t) \cup_m [\![q_2]\!]_{rel}(t)$ où $\cup_m$ est l'union multi-ensembliste

On note ici que si une requête q est bien formée, la sorte des n-uplets du résultat de l'évaluation de q est la sorte de la requête q.

Dans la formalisation de cette algèbre relationnelle proposée par la bibliothèque Coq `Datacert` que je présente dans la **Section 3.3**, les opérateurs de la projection et du renommage sont rassemblés un seul opérateur (que je nomme ici η) qui effectue d'abord une projection suivie d'un renommage. La sémantique est donc pour cette bibliothèque modifiée en :

$$[\![\eta_r(q)]\!]_{rel} = [\![\rho_r(\pi_{Ent(r)}(q))]\!]_{rel}$$

où r est une fonction de renommage.

On retombe sur les sémantiques des opérateurs précédents :

- pour la projection sur l'ensemble A, avec la fonction de renommage identité sur l'ensemble A

— pour le renommage par r, en étendant r si besoin par l'identité sur les attributs de la sorte de la sous-requête non présents dans $Ent(r)$

On remarque que la projection s'effectue sur l'ensemble des entrées de la fonction r.

Nous avons fixé ici la syntaxe et la sémantique des requêtes pour l'algèbre relationnelle positive. Cependant, certaines formalisations proposées dans les chapitres suivants étendent leur expressivité au delà de ce cadre. Dans la section suivante, nous explorons deux extensions possibles et leur sémantique. Ces extensions permettent de s'approcher d'une expressivité plus proche de celle du langage de requêtes SQL.

3.2 . Extension de l'expressivité des requêtes

Les extensions de l'expressivité des requêtes sont au nombre de deux. La première propose d'étendre la sémantique des opérateurs de la projection, du renommage et de la sélection avec des expressions arithmétiques sur les n-uplets. La deuxième élargit encore davantage l'expressivité des requêtes avec des fonctions d'agrégation.

3.2.1 . Expressions arithmétiques sur les attributs

Dans cette sous-section, la sémantique est étendue pour permettre d'effectuer des calculs arithmétiques entre les valeurs d'un même n-uplet.

Le calcul est formé par l'application d'opérations arithmétiques aux attributs et à d'éventuelles constantes. Dans la **Table 3.11**, on a appliqué à chaque n-uplet de la **Table 3.10** l'expression $A \times 2 - B + 6$. Le premier n-uplet t_1 de la **Table 3.10** devient le n-uplet t_4 en calculant le résultat de $val_A(t_1) \times 2 - val_B(t_1) + 6$: la valeur obtenue est donc $-713 \times 2 - 62 + 6 = -1482$. L'expression arithmétique désigne la valeur (par exemple $val_A(t_1)$) en indiquant l'attribut à prendre (ici A). De plus, la colonne résultante est renommée avec un nom d'attribut plus simple que l'expression arithmétique. L'attribut associé à la valeur -1482 pour le n-uplet t_4 est l'attribut C. Nous nous restreignons dans cette thèse aux opérations

	A	B
t_1	-713	62
t_2	12	81
t_3	10	21

Table 3.10 – relation r_3

	C
t_4	-1482
t_5	-51
t_6	5

Table 3.11 – Résultat de la requête appliquant l'expression arithmétique $A \times 2 - B + 6$ et la colonne est renommé en C

arithmétiques comme l'addition, la multiplication, la soustraction et l'opposé et les constantes. Une autre restriction est que les attributs sur lesquels les opérations s'appliquent doivent avoir comme domaine l'ensemble des entiers relatifs (noté $\mathbb{Z}$).

Cette extension impacte trois opérateurs : la projection, le renommage et la sélection. L'application de calculs arithmétiques sur une relation se trouve à cheval entre l'opérateur de projection et de renommage. C'est l'une des raisons pour lesquelles nous avons introduit l'opérateur η dans la section précédente. Pour résoudre ce dilemme, nous proposons ici de réunir les deux opérateurs sous un seul opérateur que l'on nommera pour la suite l'opérateur de projection sur expression. L'opérateur noté η_r^{expr} prend comme argument r une fonction de renommage étendue aux expressions d'attributs en entrée. Une expression arithmétique en entrée de r est appliquée à chaque n-uplet et les résultats sont regroupés dans la colonne ayant pour nom l'attribut associé en sortie. Pour la sélection, le changement se fait au niveau du calcul des prédicats dont les termes peuvent avoir des expressions arithmétiques : on souhaite pouvoir comparer des valeurs issues d'expression arithmétique. On note pour la suite l'opérateur de la sélection pour cette extension σ_f^{expr}.

Langage des expressions d'attributs

On définit le langage des expressions d'attributs :

$$expr^{attr} ::= c | a | expr^{attr} + expr^{attr} | expr^{attr} - expr^{attr} | expr^{attr} \times expr^{attr} | - expr^{attr}$$

où c est une constante et a est un attribut.

Une fonction de renommage a donc maintenant la syntaxe suivante :

$$r^{attr} ::= \varnothing | expr^{attr} := a | r^{attr}, r'^{attr}$$

Sorte des requêtes

La fonction définissant la sorte d'une requête $\mathbf{Attr_q}(\cdot)$ reste inchangée pour la sélection mais diffère pour la projection sur expression. La sorte de la projection sur expression est ici exactement l'ensemble des attributs de sortie de la fonction de renommage.

Définition 3.1: Sorte (Expressions d'attributs)

Soit q une requête, $e_1, .., e_n$ des expressions d'attributs et $a_1, .., a_n$ des at-

tributs.

$$\mathbf{Attr_q}(\eta^{expr}_{e_1:=a_1,..,e_n:=a_n}(q)) = \{a_1, .., a_n\}$$

Bonne formation

Pour la sélection, la bonne formation des formules est étendue avec des termes qui peuvent être des expressions d'attributs et le reste ne change pas.

En ce qui concerne la projection sur expression, on remarque ici que chaque attribut de l'ensemble de sortie de la fonction de renommage correspond à une colonne de la table résultante. Cela s'oppose aux sortes des opérateurs de projection et de renommage de l'algèbre relationnelle qui permettaient d'avoir des attributs en sortie qui n'apparaissaient pas dans la table résultante. Il est donc nécessaire que chaque renommage soit applicable sur les n-uplets de la sous-requête, c'est-à-dire que les attributs apparaissant en entrée soient dans la sorte de la sous-requête.

Pour la bonne formation de l'opérateur projection sur expression, on a donc besoin que :

- les attributs contenus dans les entrées de la fonction de renommage doivent être dans la sorte de la sous-requête
- les attributs en sortie doivent être distincts deux à deux
- pour les expressions en entrée n'étant pas un attribut, leur domaine est l'ensemble des entiers relatifs.

Définition 3.2: Bonne formation (Expressions d'attributs)

On note q une requête, $e_1, .., e_n$ des expressions d'attributs et $a_1, .., a_n$ des attributs.

$$(\!|\eta^{expr}_{e_1:=a_1,..,e_n:=a_n}(q)|\!)_{bf} = \begin{cases} (\!|q|\!)_{bf} & \text{si :} \\ & \quad \forall i, \mathbf{attr}(e_i) \subseteq \mathbf{Attr_q}(q) \\ & \quad \forall i, j, i \neq j \Rightarrow a_i \neq a_j \\ & \quad \text{tous les } e_i \text{ n'étant pas un attribut, ont} \\ & \quad \text{pour domaine } \mathbb{Z} \\ F & \text{sinon} \end{cases}$$

où $\mathbf{attr}(e_i)$ correspond aux attributs contenus dans e_i

Sémantique

L'application d'une expression e sur un n-uplet t revient à calculer la valeur obtenue en remplaçant les attributs contenus dans e par les valeurs de t pour ces attributs.

La fonction d'évaluation d'une expression d'attribut sur un n-uplet est notée $[\cdot]_e$.

Définition 3.3: Evaluation d'une expression d'attributs

Soit c une constante, a un attribut, e, e_1 et e_2 des expressions d'attributs. On suppose que chaque expression d'attributs qui suit est applicable sur la sorte de t.

- $[t_{|c}]_e = c$
- $[t_{|a}]_e = val_a(t)$ où val_a associe un n-uplet t à sa valeur pour l'attribut a
- $[t_{|e_1 * e_2}]_e = [t_{|e_1}]_e * [t_{|e_2}]_e$ où $* \in \{+, -, \times\}$
- $[t_{|-e}]_e = -[t_{|e}]_e$

On note pour la suite $a \mapsto v$, la case d'un n-uplet ayant l'attribut a et la valeur associée v.

Un n-uplet résultant de l'application d'une fonction de renommage $r = (e_1 := a_1, .., e_n := a_n)$ sur un n-uplet t est donc :

$$t[r] = (a_1 \mapsto [t_{|e_1}]_e, .., a_n \mapsto [t_{|e_n}]_e)$$

La sémantique des requêtes est modifiée seulement pour l'opérateur de projection et de la sélection. On garde la même sémantique pour les autres requêtes. Pour la sélection, la différence porte sur l'évaluation des formules. Cette évaluation est notée $[\cdot]_f^{expr}$.

Définition 3.4: Sémantique (Expressions d'attributs)

Soit q une requête et r une fonction de renommage. On suppose que les requêtes ci-après sont bien formées.

- $[\![\eta_r^{expr}(q)]\!]_{rel} = \{\{(t : \sum_{t'|t'[r]=t} \mathbf{nb}([\![q]\!]_{rel}, t'))|\exists(t' : \_) \in [\![q]\!]_{rel}, t = t[r]\}\}$
- $[\![\sigma_f^{expr}(q)]\!]_{rel} = \{\{(t : n)|\exists(t : n) \in [\![q]\!]_{rel}(t), [f]_f^{expr}(t) = V\}\}$

3.2.2 . Fonctions d'agrégation

La deuxième extension est d'étendre la sémantique avec des fonctions appliquées non plus sur un n-uplet mais sur une colonne. Ces fonctions sont les fonctions d'agrégation et dans notre contexte, nous avons la possibilité de faire la somme des valeurs d'une colonne, trouver le minimum ou le maximum parmi elles ou encore compter le nombre de valeurs qui apparaissent dans une colonne. Le calcul à appliquer est désigné par le nom de la fonction et le nom de la colonne.

Par exemple, la **Table 3.13** est la table résultante de l'application de la fonction d'agrégation "Somme A" sur la **Table 3.12**. Comme pour la projection, le renommage est possible : l'attribut associé à la fonction d'agrégation est nommé "Somme". Le n-uplet est constitué seulement de cet attribut et la valeur est égale à la somme des valeurs présentes dans la **Table 3.12**. La **Table 3.13** n'est composée que d'un seul n-uplet ayant pour attribut "Somme" et dont la valeur est le

	A	B	C
t_1	18	Vrai	1
t_2	37	Faux	6
t_3	5	Faux	8
t_4	29	Vrai	10

Table 3.12 – relation r_4

Somme
89

Table 3.13 – Somme$_A(r_4)$ renommé "Somme"

Min
5

Table 3.14 – Minimum$_A(r_4)$ renommé "Min"

Max	B
29	Vrai
37	Faux

Table 3.15 – [Maximum$_A(r_4), B$] groupé sur B renommé "Max"

Compte	B
2	Vrai
2	Faux

Table 3.16 – [Compte$_A(r_4), B$] groupé sur B renommé "Compte"

résultat de la somme des valeurs contenues dans la colonne A de la table source. De manière analogue, pour la **Table 3.14** qui applique la fonction d'agrégation "Minimum A" sur la **Table 3.12**, son seul n-uplet a pour attribut "Min" et la valeur pour cet attribut est la valeur minimale parmi les valeurs de la colonne A.

Il est possible de grouper des n-uplets selon un ensemble d'expressions d'attributs A préalablement avant d'appliquer une fonction d'agrégation. La table initiale est divisée selon les valeurs que les n-uplets ont sur l'ensemble A. Les n-uplets d'un même groupe ont les mêmes valeurs associées à chaque attribut de A. La fonction d'agrégation est ensuite appliquée non plus sur la colonne entière mais sur chaque groupe de n-uplets individuellement.

Prenons les **Tables 3.15 et 3.16**. Les n-uplets de la table de départ (**Table 3.12**) sont tout d'abord regroupés selon leurs valeurs sur l'attribut B. Il y a donc deux groupes : les n-uplets qui ont la valeur "Vrai" dans la colonne B (t_1 et t_4) et les n-uplets qui ont la valeur "Faux" (t_2 et t_3). La fonction d'agrégation est dans un deuxième temps appliquée sur chacun des groupes séparément. Cela nous donne deux n-uplets ayant pour l'attribut "Max", le maximum de chacun des groupes, 29 et 37 respectivement dans la **Table 3.15**. Pour la **Table 3.16**, pour l'attribut "Compte", comme il y a deux n-uplets dans chacun des groupes, 2 et 2 respectivement.

On autorise ici d'avoir plusieurs colonnes dans la table résultante lorsqu'on applique une fonction d'agrégation. S'il n'y a pas de groupement, les colonnes ne peuvent être que les résultats de fonctions d'agrégation. S'il y a un groupement gb, les colonnes sont :

— soit le résultat de l'application de fonctions d'agrégation après le groupement gb
— soit le résultat de l'application d'expression d'attributs contenu dans gb sur un n-uplet de chaque groupe. Cette valeur est commune pour chaque groupe.

Les autres expressions d'attributs n'ayant pas forcément de valeur commune ne peuvent pas être appliquées dans ce cas. Par exemple, pour les **Tables 3.15 et 3.16**, l'attribut B est accepté car c'est l'attribut sur lequel on fait le groupement. La valeur correspondant à l'attribut B pour chaque groupe est la même. A l'inverse, l'attribut C ne peut être accepté. Il y a deux valeurs possibles pour le groupe avec l'attribut "B = Vrai" (1 et 10) et il n'est pas possible de faire un choix entre les deux.

L'ajout des fonctions d'agrégation n'impacte pas les autres opérateurs et demande l'introduction d'un nouvel opérateur spécifique $\Gamma_{Eagg,gb}$. $Eagg$ désigne les fonctions d'agrégation et des expressions d'attribut avec renommage qui constituent les colonnes de la table résultante. gb désigne le groupement éventuel à effectuer et est constitué d'un ensemble d'expressions d'attributs.

Langage des fonctions d'agrégation

La syntaxe du langage agg des fonctions d'agrégation est la suivante :

$$agg ::= \text{Somme } a|\text{Compte } a|\text{Min } a|\text{Max } a$$

où a est un attribut.

Sorte des requêtes

La fonction de la sorte d'une requête $\mathbf{Attr}_{\mathsf{q}}(\cdot)$ est étendue de manière analogue à celle de la projection avec expressions d'attributs : les attributs de la requête sont ceux donnés au niveau du renommage.

Définition 3.5: Sorte (Fonctions d'agrégation)

Soit q une requête, gb un ensemble fini d'expressions d'attributs, $e_1, .., e_n$ des fonctions d'agrégation ou d'expression d'attributs et $a_1, .., a_n$ des attributs.

$$\mathbf{Attr}_{\mathsf{q}}(\Gamma_{(e_1:=a_1,..,e_n:=a_n),gb}(q)) = \{a_1, .., a_n\}$$

Bonne formation

L'évaluation d'une fonction d'agrégation $agg(a)$ sur une relation r est possible seulement si l'attribut a est dans la sorte de r. Dans cette thèse, nous limitons

les attributs pris en argument des fonctions d'agrégation "Somme", "Minimum" et "Maximum" aux attributs dont le domaine est inclus dans l'ensemble des entiers relatifs. La fonction d'agrégation "Compte" reste applicable sur n'importe quel domaine. Le groupement selon une expression d'attributs e sur une relation r ne peut se faire que si les attributs apparaissant dans e sont dans la sorte de r.

Pour les expressions d'attribut e dans $Eagg$, e doit faire partie de l'ensemble selon lequel on groupe.

De nombreux critères nécessaires pour l'application de l'opérateur Γ ont déjà été énoncés plus haut et sont repris dans la définition de la bonne formation pour celui-ci. Les autres critères sont analogues à ceux de la projection pour le renommage pour les mêmes raisons.

Définition 3.6: Bonne formation (Fonctions d'agrégation)

Soit q une requête, gb un ensemble fini d'expressions d'attributs, $e_1, .., e_n$ des fonctions d'agrégation ou des expressions d'attributs et $a_1, .., a_n$ des attributs.

$$(\!|\Gamma_{(e_1:=a_1,..,e_n:=a_n),gb}(q)|\!)_{bf} = \begin{cases} (\!|q|\!)_{bf} & si : \\ & \quad \forall i, \mathbf{attr}(e_i) \subseteq \mathbf{Attr_q}(q) \\ & \quad \forall g \in gb, \mathbf{attr}(g) \subseteq \mathbf{Attr_q}(q) \\ & \quad \forall i, j, i \neq j \Rightarrow a_i \neq a_j \\ & \quad \forall i, e_i \in expr^{attr} \Rightarrow e_i \in gb \\ & \quad \forall i, a, \\ & \quad\ (e_i \in \{\text{Min } a, \text{Max } a, \text{Somme } a\} \\ & \quad\ \Rightarrow dom(a) \in \mathbb{Z}) \\ F & sinon \end{cases}$$

où $dom(a)$ donne le domaine associé à l'attribut a

Sémantique

Nous commençons en établissant l'évaluation d'une fonction d'agrégation sur une seule relation.

On note $[\cdot]_{fagg}$ la fonction d'évaluation d'une fonction d'agrégation sur une relation.

Définition 3.7: Evaluation d'une fonction d'agrégation

Soit r une relation non vide et a des attributs. On suppose que a est dans la sorte de r.

- $[\text{Compte } a]_{fagg}(r) = \sum_{(t:n)\in r} n$
- $[\text{Somme } a]_{fagg}(r) = \sum_{(t:n)\in r} [t_{|a}]_e \times n$ si $dom(a) = \mathbb{Z}$

— [Min a]$_{fagg}(r) = min\{x \in \mathbb{Z} | \exists (t : n) \in r, [t_{|a}]_e = x\}$ si $dom(a) = \mathbb{Z}$
— [Max a]$_{fagg}(r) = max\{x \in \mathbb{Z} | \exists (t : n) \in r, [t_{|a}]_e = x\}$ si $dom(a) = \mathbb{Z}$

On remarque que nous appliquons bien les fonctions d'agrégation sur toute une colonne de la relation.

Nous définissons ensuite l'évaluation de potentiellement plusieurs fonctions d'agrégation et d'expression d'attributs avec renommage.

Définition 3.8: Evaluation sur une relation avec renommage

Soit r une relation non vide, f une fonction d'agrégation, e une expression d'attributs, a, a_1, .., a_n des attributs et fe_1, .., fe_n des expressions d'attributs ou des fonctions d'agrégation.
On suppose que pour toutes les expressions d'attributs dans l'ensemble $\{fe_1, .., fe_n\}$, On suppose que les a_i sont deux à deux distincts et que les attributs dans f, e, fe_1,.., fe_n sont dans la sorte de r.

— $[\![r]\!]_{agg}(f := a) = a \mapsto [f]_{fagg}(r)$
— $[\![r]\!]_{agg}(e := a) = a \mapsto [t_{|e}]_e$ où t est un n-uplet de r.
— $[\![r]\!]_{agg}(fe_1 := a_1, .., fe_n := a_n) =$
$$([\![r]\!]_{agg}(fe_1 := a_1), .., [\![r]\!]_{agg}(fe_n := a_n))$$

Pour passer à l'évaluation de fonctions d'agrégation sur des groupes, nous définissons ensuite la fonction $Group(\cdot, \cdot)$ qui découpe en groupe la table résultante d'une requête selon un ensemble d'expression d'attributs. Chaque groupe est composé de l'ensemble des n-uplets ayant les mêmes valeurs après évaluation sur un ensemble d'expression d'attributs gb.

Définition 3.9: Groupement selon gb

Soit q une requête et gb un ensemble fini d'expressions d'attributs.
On suppose la requête bien formée et que les attributs contenus dans gb sont dans la sorte de q.

$$Group(q, gb) = \{r \subseteq [\![q]\!]_{rel} | \exists r, \exists (t : n) \in r, \forall (t', n') \in [\![q]\!]_{rel}, \\ (t', n') \in r \Leftrightarrow [t'_{|gb}]_e = [t_{|gb}]_e$$

On pose $nb_{rel}(Gr, t, (fe_1 := a_1, .., fe_n := a_n))$ la fonction qui pour un ensemble de relations Gr renvoie le nombre d'éléments de Gr pour lesquels leur évaluation par $[\![\cdot]\!]_{agg}(fe_1 := a_1, .., fe_n := a_n)$ donne t. Cela nous donne le nombre d'occurrence de t lors de l'évaluation de Gr par les fonctions d'agrégations et des expressions d'attributs.

A partir des groupes, nous définissons la sémantique de l'opérateur Γ. Le nombre de fois où un n-uplet apparaît dépend ici du nombre de fois où l'évaluation d'un groupe donne le même résultat.

Définition 3.10: Sémantique (Fonctions d'agrégation)

Soit q une requête, gb un ensemble fini d'expression d'attributs, $fe_1, .., fe_n$ des fonctions d'agrégation ou des expressions d'attributs et $a_1, .., a_n$ des attributs.
On suppose les requêtes bien formées.

$$[\![\Gamma_{fe_1:=a_1,..,fe_n:=a_n,gb}(q)]\!]_{rel} = \{\{t : nb_{rel}(Gr, t, (f_1 := a_1, .., fe_n := a_n)) | \exists r \in Group([\![q]\!]_{rel}, gb), t = [\![r]\!]_{agg}(fe_1 := a_1, .., fe_n := a_n)\}\}$$

On remarque que le cas où il n'y a pas de groupement est inclus par la fonction $Group$ quand gb est l'ensemble vide. Cependant, cette sémantique ne correspond pas totalement à ce qu'on a énoncé précédemment ou celle du langage SQL. En effet, lorsque la table associée à la sous-requête est vide et il n'y a pas de groupement, le résultat de l'application d'une fonction d'agrégation est 0 si c'est la fonction "Compte" et la valeur Null sinon. Cependant la bibliothèque `Datacert` a comme spécification que dans ce cas précis, la table renvoyée est vide. La sémantique présentée correspond à celle de `Datacert`. Cette différence étant en cours de modification dans `Datacert`, ce cas limite est dans la version actuelle des formalisations laissé comme précisé ci-dessus et sera modifié à une date ultérieure.

Nous avons posé la syntaxe, la bonne formation et la sémantique du langage de requêtes sur lequel s'est porté notre choix pour le cadre de cette thèse. Dans la partie suivante, nous introduisons la bibliothèque `Datacert` qui propose une formalisation dans l'assistant de preuve Coq de l'algèbre relationnelle et des deux extensions présentées. Nous expliquons aussi la représentation abstraite des requêtes et leur évaluation.

3.3 . Formalisation de l'algèbre relationnelle : la bibliothèque Datacert

La bibliothèque `Datacert` [23, 26, 27] formalise dans l'assistant de preuves `Coq` les structures de données, la syntaxe et la sémantique de l'algèbre relationnelle étendue aux fonctions d'agrégation et aux expressions d'attributs. Elle contient aussi un compilateur certifié permettant de passer de SQL vers une algèbre relationnelle encore plus étendue et propose une preuve d'équivalence entre la sémantique de cette algèbre et SQL. Elle gère aussi les valeurs NULL pouvant apparaître dans les tables issues du langage SQL.

La particularité de cette bibliothèque est que les fonctions principales pour évaluer des requêtes sont exécutables et se déclinent en plusieurs niveaux d'abstraction. La bibliothèque s'articule en deux temps : une partie abstraite et une partie instanciée.

L'avantage d'avoir cette dualité repose sur le fait que nous avons une structure suffisamment abstraite pour laisser à l'utilisateur le choix d'implémenter les fonctions et les structures de la manière qu'il lui convient le mieux, d'ajouter des optimisations ainsi qu'en ayant la garantie que celles-ci vérifient bien les propriétés d'intérêt. De plus, la partie instanciée permet d'avoir une implémentation capable d'exécuter des requêtes et d'obtenir des résultats.

Une structure adéquate pour cette dualité et utilisée dans cette bibliothèque est les enregistrements. Cette structure permet de regrouper les types et les fonctions associées nécessaires à la description d'un objet d'une structure de données et évite d'implémenter l'objet totalement. L'assistant de preuve ajoute la possibilité de raffiner le comportement des types et des fonctions à l'aide de propriétés qui doivent être vérifiées pour être une instance de ces enregistrements.

Dans cette section, nous présentons ces deux niveaux d'abstraction en commençant par la représentation abstraite des données. Dans la première sous-section, nous décrivons les enregistrements formalisant les relations d'ordre total très utilisées dans cette bibliothèque ainsi que ceux qui formalisent les structures de données, les requêtes, la sorte de celles-ci et leur évaluation. La seconde sous-section présente les types et fonctions permettant d'instancier et d'exécuter des structures de données ainsi que l'instanciation de la fonction d'évaluation.

3.3.1 . Représentation abstraite des données

Relations d'ordre total

La bibliothèque `Datacert` travaille sur de nombreuses structures comportant une relation d'ordre total et décidable. Nous présentons donc ici ses formalisations.

Les relations d'ordre $\leqslant$ sur un ensemble A sont ici définies à partir d'une fonction de comparaison $compare$ qui leur est associée. Cette fonction renvoie un élément du type `comparison` de la bibliothèque standard qui se décompose en égalité `Eq`, inférieur strictement `Lt` et supérieur strictement `Gt`.

La bibliothèque définit donc l'enregistrement `Oeset.Rcd` qui renferme la fonction de comparaison liée à une relation d'ordre total.

```
Module Oeset.
Record Rcd (A : Type) : Type :=
mk_R
  {
    compare : A → A → comparison;
    compare_eq_trans :
      ∀ a1 a2 a3, compare a1 a2 = Eq → compare a2 a3 = Eq →
      compare a1 a3 = Eq;
    compare_eq_lt_trans :
      ∀ a1 a2 a3, compare a1 a2 = Eq → compare a2 a3 = Lt →
      compare a1 a3 = Lt;
    compare_lt_eq_trans :
      ∀ a1 a2 a3, compare a1 a2 = Lt → compare a2 a3 = Eq →
```

```
      compare a1 a3 = Lt;
    compare_lt_trans :
      ∀ a1 a2 a3, compare a1 a2 = Lt → compare a2 a3 = Lt →
      compare a1 a3 = Lt;
    compare_lt_gt : ∀ a1 a2, compare a1 a2 = CompOpp (compare a2 a1)
  }.
```

où `CompOpp` renvoie `Eq` si son argument est `Eq`, `Gt` s'il est `Lt` et `Lt` sinon.

L'enregistrement se décompose en la fonction `compare` ainsi que les propriétés qu'elle doit vérifier pour assurer que l'ordre soit bien total. Le caractère réflexif et symétrique est une conséquence de `compare_lt_gt` et le caractère transitif est assuré par l'hypothèse `compare_lt_trans`. L'enregistrement ajoute quelques propriétés supplémentaires : les hypothèses supplémentaires sont des suppositions classiques sur les inégalités mais qu'il est nécessaire de mettre en lumière lorsque l'on utilise un assistant de preuve. On remarque bien que la fonction `compare` impose le caractère total à la relation d'ordre de par sa définition.

D'autres structures découlent de l'enregistrement `Oeset.Rcd` : les ensembles finis d'éléments avec `Feset.Rcd` et les multi-ensembles finis d'éléments avec `Febag.Rcd` sont constitués d'éléments d'un ensemble possédant une relation d'ordre associée au type `Oeset.Rcd`. Ces deux structures sont définies sous la forme d'enregistrement abstrait et proposent différentes opérations entre leurs éléments que nous détaillerons lorsqu'elles seront introduites.

L'enregistrement `Oset.Rcd` est similaire à `Oeset.Rcd` mais la différence repose sur la fonction de comparaison qui coïncide pour l'égalité avec l'égalité syntaxique de Coq. De même que `Feset.Rcd` avec `Oeset.Rcd`, l'enregistrement `Fset.Rcd` propose des ensembles finis d'éléments dont une relation d'ordre est de type `Oset.Rcd`.

Représentation des structures de données

L'enregistrement suivant permet la représentation des structures de données dans `Datacert`. Les premiers types de l'enregistrement permettent de définir les différents éléments nécessaires à une structure de données :

- `type` : les noms des domaines possibles pour les attributs
- `attribute` : l'ensemble des attributs possibles
- `value` : l'ensemble des valeurs
- `predicate` : l'ensemble des prédicats
- `symbol` : l'ensemble des symboles des expressions d'attributs
- `aggregate` : l'ensemble des fonctions d'agrégation
- `B` : l'ensemble des valeurs de retour des prédicats

Ces types sont pour la plupart associés à une relation d'ordre et sont abstraits. Les fonctions `type_of_attribute` et `type_of_value` attribuent un domaine pour chaque attribut et pour chaque valeur respectivement. Les prédicats, les symboles et les fonctions d'agrégation sont applicables sur des valeurs par l'inter-

médiaire des fonctions d'interprétation : `interp_predicate`, `interp_symbol` et `interp_aggregate`. On note que les valeurs ont une valeur par défaut pour chaque type `default_value`.

```
Module Tuple.
Record Rcd : Type :=
 mk_R
   {
   type : Type;
   attribute : Type;
   value : Type;
   predicate : Type;
   symbol : Type;
   aggregate : Type;
   B : Bool.Rcd;
   (** Interpretation *)
   interp_predicate : predicate → list value → Bool.b B;
   interp_symbol : symbol → list value → value;
   interp_aggregate : aggregate → list value → value;
   (** particular values *)
   default_value : type → value;
   (** comparisons *)
   OAtt : Oset.Rcd attribute;
   A : Fset.Rcd OAtt;
   OVal : Oset.Rcd value;
   OPred : Oset.Rcd predicate;
   OSymb : Oset.Rcd symbol;
   OAgg : Oset.Rcd aggregate;
   (** Typing attributes and values. *)
   type_of_attribute : attribute → type;
   type_of_value : value → type;
   (** Abstract tuples *)
   tuple : Type;
   OTuple : Oeset.Rcd tuple;
   (** labels and field extraction *)
   labels : tuple → Fset.set A;
   dot : tuple → attribute → value;
   mk_tuple : Fset.set A → (attribute → value) → tuple;
   (** Properties *)
   labels_mk_tuple : ∀ s f, labels (mk_tuple s f) ≡ s;
   dot_mk_tuple :
     ∀ a s f, dot (mk_tuple s f) a =
                  if a ∈? s then f a else default_value (type_of_attribute a);
   tuple_eq :
     ∀ t1 t2, (Oeset.compare OTuple t1 t2 = Eq) ⟷
          (labels t1 ≡ labels t2 ⋀ ∀ a, a ∈ (labels t1) → dot t1 a = dot t2 a)
 }.
```

La deuxième partie de l'enregistrement est principalement focalisée sur la définition des n-uplets avec le type `tuple` et les propriétés qu'ils doivent vérifier. Pour permettre la comparaison entre deux n-uplets, le type `tuple` possède une relation d'ordre `OTuple`. La fonction `labels` donne accès à l'ensemble des attributs d'un

n-uplet particulier et `dot` indique la valeur d'un n-uplet pour un attribut donné. La création d'un n-uplet se fait avec `mk_tuple` en donnant en argument l'ensemble des attributs qu'on souhaite et une fonction qui associe, pour chaque attribut, la valeur que prend ce n-uplet pour cet attribut. Pour assurer que l'application de `labels` et `dot` au n-uplet créé par `mk_tuple` renvoie les résultats attendus, l'enregistrement contient aussi les propriétés `labels_mk_tuple` et `dot_mk_tuple` que chaque instance doit vérifier. De même, la propriété `tuple_eq` définit le lien entre l'équivalence entre deux n-uplets et les résultats obtenus en appliquant `labels` et `dot` à ces deux n-uplets.

Représentation abstraite des requêtes et de leur évaluation

Ayant maintenant une structure de données, nous définissons les requêtes et leur évaluation. Dans ce qui suit, `T` est une instance de l'enregistrement `Tuple.Rcd`.

Les types proposés pour les formules et les requêtes ici sont une restriction des types proposés par la bibliothèque `Datacert`. Cette restriction présentée dans ce chapitre est celle utilisée pour le chapitre suivant. Dans les **Chapitres 5 et 6**, l'expressivité n'est pas exactement la même et je clarifierai alors sur quelle restriction nous nous trouvons.

Le type des formules `formula` est défini inductivement en reprenant les éléments énoncés dans la **Sous-section 3.1.1**.

```
Inductive formula : Type :=
| Sql_Conj : and_or → formula → formula → formula
| Sql_Not : formula → formula
| Sql_True : formula
| Sql_Pred : Tuple.predicate T → list (@funterm T) → formula.
```

Le type `and_or` désigne soit le "et" soit le "ou" logique. Le type `Tuple.predicate T` est un des types abstraits définis par l'instance `T` et correspond à l'ensemble des prédicats possibles. `@funterm T` est le type des expressions d'attributs : à ce niveau d'abstraction les symboles sont abstraits et proviennent de `T`. Étant au niveau abstrait, on laisse la possibilité d'appliquer un prédicat sur plus ou moins que deux expressions d'attributs. Dans les cas concrets, il n'est possible d'appliquer des prédicats binaires.

Le langage des requêtes présenté ici correspond à celui contenant les expressions d'attributs et les fonctions d'agrégation. Il est représenté sur forme d'un type inductif `query`.

```
Inductive query : Type :=
  | Q_Empty_Tuple : query (* n–uplet vide *)
  | Q_Table : relname → query (* table *)
  | Q_Union : query → query → query (* union *)
  | Q_NaturalJoin : query → query → query (* jointure naturelle *)
  | Q_Pi : list (select T) → query → query (* projection sur expression*)
  | Q_Sigma : formula → query → query (* selection avec expression *)
  | Q_Gamma : list (select T) → list funterm →
```

```
        query → query. (* fonction d'agregation *)
```

Le type `relname` désigne l'ensemble des noms des relations sources. Pour le cas des fonctions d'agrégation, le type `@funterm T` désigne le groupement à effectuer. Le type `list (select T)` est le type des expressions d'attributs et des fonctions d'agrégation avec renommage.

```
Inductive select : Type :=
    Select_As : aggterm → attribute → select
```

`aggterm` est le type qui regroupe les fonctions d'agrégation et les expressions d'attributs. Dans le cas du type `select`, il désigne les calculs à appliquer et le deuxième élément donne l'attribut que l'on souhaite donner à la colonne au niveau du renommage. On remarque que la projection peut ainsi prendre en argument des fonctions d'agrégation. C'est le rôle de la fonction de bonne formation d'éliminer ces cas indésirables en renvoyant que la requête est mal formée.

On décrit ensuite deux des fonctions présentées précédemment : la sorte d'une requête et sa sémantique. La fonction de bonne formation étant plus complexe dans la bibliothèque `Datacert` que celles dont nous avons parlé plus tôt, nous y reviendrons dans les chapitres qui suivent.

La fonction `sort` est une fonction récursive sur sa structure qui associe à une requête sa sorte. Le constructeur `Fixpoint` indique son caractère récursif et le type de sortie est `setA` qui est le type des ensembles finis d'attributs.

```
Fixpoint sort (q : query) : setA :=
  match q with
    | Q_Empty_Tuple ⇒ Fset.empty
    | Q_Table t ⇒ basesort t
    | Q_Set q1 _ ⇒ sort q1
    | Q_NaturalJoin q1 q2 ⇒ Fset.union (sort q1) (sort q2)
    | Q_Sigma _q ⇒ sort q
    | Q_Pi l _
    | Q_Gamma l ___⇒
          Fset.mk_set (map (fun x ⇒ match x with Select_As _a ⇒ a end) l)
  end.
```

La fonction `basesort` donne l'ensemble d'attributs associé à une table source. `Fset.empty` est l'ensemble vide et `Fset.union` effectue l'union ensembliste de deux ensembles. Pour la projection et les fonctions d'agrégation, on récupère à l'aide de la fonction `map` les deuxièmes éléments du type `select` (qui sont des attributs avec lesquels on veut renommer les colonnes) puis on convertit la liste obtenue en un ensemble grâce la fonction `Fset.mk_set`.

L'évaluation d'une requête s'effectue à l'aide de la fonction inductive sur sa structure `eval_query`. Elle prend en argument un environnement `env` et une requête. Le type des éléments renvoyés est `bagT` qui correspond aux multi-ensembles finis de n-uplets (instance de la bibliothèque `Febag.Rcd`).

```
Fixpoint eval_query env q {struct q} : bagT :=
  match q with
```

```
| Q_Empty_Tuple ⇒ Febag.singleton _empty_tuple
| Q_Table r ⇒ instance r
| Q_Set o q1 q2 ⇒
  if sort q1 ≡? sort q2
  then Febag.union _(eval_query env q1) (eval_query env q2)
  else Febag.empty _
| Q_NaturalJoin q1 q2 ⇒ natural_join_bag (eval_query env q1) (eval_query env q2)
| Q_Pi s q ⇒
  Febag.map __(fun t ⇒ projection (env_t env t) s) (eval_query env q)
| Q_Sigma f q ⇒
  Febag.filter
    _(fun t ⇒ Bool.is_true (B T) (eval_sql_formula (env_t env t) f))
    (eval_query env q)
| Q_Gamma s lf q ⇒
  Febag.mk_bag
    _(map (fun l ⇒ projection (env_g env (Group_By lf) l) (Select_List s))
          (make_groups env (eval_query env q) (Group_By lf))))
end.
```

Les environnements sont utilisés pour les opérateurs de projection et d'agrégat principalement dans le cadre des requêtes imbriquées. La bibliothèque `Datacert` permet d'effectuer ce dernier type de requête mais notre restriction ne les inclut pas.
Les environnements stockent en mémoire des triplets contenant les informations suivantes : une liste de n-uplets, les attributs associés aux n-uplets et si on effectue un groupement, le cas échéant, les expressions d'attributs du groupement. Les fonctions `env_t` et `env_g` ajoutent ainsi dans un environnement les n-uplets (pour la projection, le n-uplet sur lequel on projette et pour les fonctions d'agrégation, un des groupes de n-uplets) et les groupements à effectuer (avec le constructeur `Group_By`) ou non. L'environnement est ensuite utilisé par la fonction `projection` pour effectuer les calculs projetant un n-uplet ou appliquant une fonction d'agrégation sur un groupe. Pour plus d'informations sur le fonctionnement de l'environnement pour `Datacert` dans des cas plus complexes : [23, 26, 27]. Nous gardons dans cette thèse cette structure car les fonctions effectuant les projections et les applications de fonctions d'agrégation les utilisent et nous souhaiterions à plus long terme étendre les sémantiques que je propose dans les prochains chapitres aux requêtes imbriquées.

La fonction `instance` associe à chaque nom de relation sa table correspondante. `empty_tuple` est le n-uplet vide <>. Nous ne détaillons pas la fonction `natural_join_bag` ; elle renvoie le multi-ensemble de la jointure naturelle des deux sous requêtes. La fonction `projection` applique les expressions d'attributs et les fonctions d'agrégation avec renommage sur l'environnement. Le découpage en groupes de n-uplets selon `lf` est effectué avec la fonction `make_groups`.

La bibliothèque `Febag` propose les fonctions suivantes :

— `Febag.singleton` qui crée un multiensemble avec l'élément en argument, ici le n-uplet vide

— `Febag.map` qui applique à chaque élément d'un multi-ensemble une fonction. Dans notre cas, il projette les n-uplets sur l'ensemble d'attributs `s`
— `Febag.filter` qui garde les éléments d'un multi-ensemble vérifiant une formule. La formule ici est passée en argument à la fonction `eval_sql_formula` qui l'évalue sur un n-uplet qui est stocké dans l'environnement
— `Febag.mk_bag` qui crée un multi-ensemble à partir d'une liste. Cette fonction est utilisée car la fonction `make_groups` renvoie des listes et non des multi-ensembles.

On note que nous vérifions pour l'union entre deux multi-ensembles que les sortes des sous-requêtes sont bien équivalentes.

Pour pouvoir exécuter le code produit sur des cas concrets, la partie qui suit donne les instanciations des types et les fonctions de la structure de données pour pouvoir évaluer des requêtes.

3.3.2 . Représentation concrète

Nous présentons ici les instanciations des types principaux de l'enregistrement `Tuple.Rcd`.

Représentation des structures de données

Pour les domaines des attributs, nous acceptons : les chaînes de caractères, les entiers relatifs et les booléens. Le champ `type` de l'enregistrement représente les trois domaines possibles :

```
Inductive type : Set :=
    type_string : type | type_Z : type | type_bool : type
```

Les attributs et les valeurs sont donc décomposés selon ces trois domaines.

Chaque attribut possède un identifiant (dans l'ensemble des entiers naturels formalisé par `N`) et un nom (formalisé par une chaîne de caractères `string`).

```
Inductive attribute : Set :=
  | Attr_string : N → string → attribute
  | Attr_Z : N → string → attribute
  | Attr_bool : N → string → attribute.
```

Les valeurs prennent en argument un type `option` pour pouvoir gérer les valeurs Null. Les valeurs Null sont représentées par `None` et les autres à l'aide de `Some`.

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

Les valeurs par défaut pour chaque domaine sont les valeurs Null.

Les symboles peuvent être soit :

— une opération unaire ou binaire parmi l'addition, la multiplication, la soustraction et l'opposé entre deux valeurs
— une constante.

```
Inductive symbol (value : Type) : Type :=
    Symbol : string → Values.symbol value
  | CstVal : value → Values.symbol value
```

Le type `symbol` reste abstrait pour les opérations mais la fonction `interp_symbol` évalue les symboles selon la liste ci-avant.

```
Definition interp_symbol f (l : list value) :=
    match f with
    | Symbol s ⇒
        if comparison_case (string_compare s "plus")
        then interp_binop_Z Z.add
        else
         if comparison_case (string_compare s "mult")
         then interp_binop_Z Z.mul
         else
          if comparison_case (string_compare s "minus")
          then interp_binop_Z Z.sub
          else
           if comparison_case (string_compare s "opp")
           then interp_unop_Z Z.opp
           else fun _: list value ⇒ Value_Z None
    | CstVal v ⇒
        fun l : list value ⇒
        match l with
        | nil ⇒ v
        | _:: _ ⇒ default_value (type_of_value v)
        end
    end.
```

Si le symbole évalué est une constante, la liste doit être vide et on renvoie la valeur contenue dans le symbole. La fonction `interp_binop_Z` (réciproquement `interp_unop_Z`) applique l'opération correspondante sur les deux premiers éléments (réciproquement sur l'élément de tête). Dans le cas où le symbole n'existe pas ou la liste est d'une taille inadéquate, une valeur par défaut est renvoyée.

Les types et les fonctions d'interprétations pour les prédicats et les fonctions d'agrégation suivent une construction similaire. Les prédicats et des fonctions d'agrégation acceptés sont cités dans les **Sous-sections 3.1.1 et 3.2.2**.

Pour le type B de retour des prédicats, la bibliothèque Datacert propose deux instances : les booléens classiques, et les booléens à trois valeurs offrant la possibilité d'exprimer une indécision avec la valeur unknown3.

```
Inductive bool3 : Set :=
    true3 : bool3 | false3 : bool3 | unknown3 : bool3
```

Enfin, les n-uplets sont représentés par le type `list (attribute * value)` qui associe pour chaque attribut une valeur.

Pour que l'instanciation soit possible, ces différents types et fonctions doivent vérifier les trois propriétés `labels_mk_tuple`, `dot_mk_tuple` et `tuple_eq`. Nous ne présentons pas ici les preuves de ces lemmes.

Représentation concrète d'évaluation des requêtes

Maintenant que nous avons la représentation des données, nous nous intéressons à ce qu'il reste à instancier pour pouvoir évaluer des requêtes.

L'instanciation de l'enregistrement `Tuple.Rcd` avec les éléments cités ci-avant est notée pour la suite `TNull` dans le cadre de l'algèbre étendue aux expressions d'attributs et aux fonctions d'agrégation.

Il manque pour évaluer les requêtes les tables sources. Pour cela, le type `db_state_` représente une base de données.

```
Record db_state_ T : Type :=
  mk_state
    {
      _relnames : list relname;
      _basesort : relname → Fset.set (A T);
      _instance : relname → Febag.bag (Fecol.CBag (CTuple T))
    }.
```

`_relnames` donne l'ensemble des noms des tables sources définies, `_basesort` associe à chaque table source sa sorte et `_instance` donne l'ensemble des n-uplets présents dans une table source particulière.

La fonction `eval_query` attend en arguments explicites la fonction qui donne la sorte de chaque table source, les n-uplets présents dans ces tables, la valeur d'indécision des booléens et la fonction `contains_nullsRA` qui indique si une valeur est la valeur Null ou non.

```
Notation eval_query_rel_NullRA (db : db_state_) :=
      (eval_query (_basesort db) (instance_relRA db) unknown3 contains_nullsRA).
```

3.4 . Autres bibliothèques

Dans cette section, nous abordons deux bibliothèques, que j'ai implémentées et utilisées dans les différentes formalisations présentées dans les chapitres suivants. La première bibliothèque a pour but de faciliter les preuves impliquant la fonction `fold_left`. La seconde propose une formalisation de fonctions partielles sur des entiers finis compatibles avec les équivalences.

3.4.1 . Lemmes pour la fonction fold_left

La bibliothèque `FoldFacts` a été implémentée dans le but de faciliter les preuves d'équivalences impliquant la fonction `fold_left`. Pour rappel, la fonction

```
fold_left : forall A B : Type, (A -> B -> A) -> list B -> A -> A
```

est une fonction récursive avec accumulateur. Elle prend en argument deux types A et B, une fonction f, une liste l et un accumulateur a. A chaque itération, elle applique la fonction f sur un élément de la liste et l'accumulateur, et le résultat

devient l'accumulateur pour la prochaine itération. La liste est ici parcourue de gauche à droite.

Il existe déjà des bibliothèques proposant des lemmes sur des égalités pour la fonction `fold_left`. Cependant, nous travaillons dans cette thèse avec des éléments équivalents et non égaux, ce qui rend ces lemmes difficilement utilisables. De plus, la fonction `fold_left` est difficile à manipuler dans une preuve par induction sur la liste qu'elle prend en argument. En effet, `fold_left f (hd :: tl) a` donne après simplification `fold_left f tl (f hd a)` et cela complique l'application de l'hypothèse.

Je propose d'introduire la fonction `fold_left_wacc` récursive sans accumulateur pour remédier à cela :

```
Fixpoint fold_left_wacc (l : list A) (vend : B) (g : A → B) :=
  match l with
  |nil ⇒ vend
  |hd :: tl ⇒ op (g hd) (fold_left_wacc tl vend g)
  end.
```

où `op` est une opération entre deux éléments de `B`.

Pour avoir une équivalence entre la fonction `fold_left` et la fonction `fold_left_wacc`, il est nécessaire de poser l'hypothèse `Hypothesis HCM : CM zero op OB`. Cette hypothèse suppose que (`B`, `zero`, `op`) forme un monoïde commutatif pour la relation d'équivalence `OB`. L'enregistrement `CM` sera défini dans la **Sous-section 4.2.2**.

```
Lemma fold_left_wacc_equiv: ∀ (l : list A) (g : A → B) a,
    eq (fold_left (fun x y ⇒ op (g y) x) l a) (fold_left_wacc l a g).
```

Il suffit ainsi d'utiliser l'équivalence entre `fold_left` et `fold_left_wacc` pour transformer les preuves sur `fold_left` en preuve sur `fold_left_wacc`. Le caractère "récursive sans accumulateur" de `fold_left_wacc` résout le problème rencontré sur les inductions.

La fonction `fold_left` sera utilisée dans la définition de la sémantique des types de provenances proposées dans les chapitres suivants et donc dans les preuves relatives à celles-ci. Je propose maintenant quelques uns des lemmes qui m'ont permis de manipuler plus facilement les fonctions `fold_left` du cours des preuves de cette thèse. Dans la suite de la sous-section, nous supposons que l'hypothèse `HCM` est toujours vérifiée.

Il y a par exemple le lemme suivant :

```
Lemma fold_left_wacc_eq_3 : ∀ g1 g2 l a,
    (∀ a b, eq a b → eq (g1 a) (g2 b)) →
    eq (fold_left_wacc l a g1) (fold_left_wacc l a g2).
```

qui prouve que deux fonctions `fold_left_wacc` sont équivalentes si les fonctions `g1` et `g2` qu'elles prennent en argument sont équivalentes et compatibles avec l'équivalence de A.

Le lemme `commut_fold_left_wacc_same_3` permet d'échanger l'ordre dans lequel on applique deux fonctions `fold_left_wacc`.

```
Lemma commut_fold_left_wacc_same_3 : ∀ l1 l2 g,
    eq (fold_left_wacc l1 zero
          (fun y ⇒ fold_left_wacc l2 zero (fun x ⇒ g x y)))
       (fold_left_wacc l2 zero
          (fun y ⇒ fold_left_wacc l1 zero (fun x ⇒ g y x))).
```

Cela est possible si les listes sont indépendantes par rapport aux variables des fonctions prises en argument des `fold_left_wacc`.

Un dernier lemme que j'aborderais par la suite :

```
Lemma fold_left_wacc_equiv_eq_3_2 : ∀ l f1 f2 b,
 (∀ a, f1 a = f2 a) →
  eq (fold_left_wacc l b f1) (fold_left_wacc l b f2).
```

C'est une variante du lemme `fold_left_wacc_eq_3` : les fonctions ici ne sont pas obligatoirement compatibles avec l'équivalence de A et doivent être égales pour chaque élément de A.

3.4.2 . Formalisation des fonctions partielles

La bibliothèque `FiniteMap` propose des fonctions partielles abstraites dont les ensembles de départ et d'arrivée possèdent des relations d'ordre. Ces fonctions ont aussi la particularité d'être définie sur un sous-ensemble fini. Comme pour les autres structures sous forme d'enregistrement, une instanciation concrète des fonctions partielles est proposée.

Formalisation abstraite des fonctions partielles

Les fonctions partielles, leurs propriétés et leurs opérations sont regroupées dans un enregistrement. La bibliothèque est divisée en plusieurs variantes selon les opérations proposées.

Ces variantes ont une base commune de fonctions et de types. Chaque enregistrement prend en argument les ensembles d'entrée (`Elt1`) et de sortie (`Elt2`) ainsi que leurs relations d'équivalence (`OE1` et `OE2`) respectives.

```
Record Rcd (Elt1 Elt2 : Type) (OE1 : Oeset.Rcd Elt1) (OE2 : Oeset.Rcd Elt2) :=
 mk_R
    {
      femap : Type;
      FeElt1 : Feset.Rcd OE1;
      base : femap → Feset.set FeElt1;
      find : Elt1 → femap → option Elt2;
      find_spec_1 : ∀ f x,
           x ∈ (base f) = true ⟷
           ∃y, find x f = Some y;
      find_spec_2 : ∀ f x x',
           eq x x' → find x f = find x' f;
      ...
```

Le type des fonctions partielles est `femap`. L'ensemble des ensembles finis d'éléments est défini par le type `FeElt1` ce qui permet de restreindre les fonctions partielles sur un ensemble fini de `Elt1`.

Chaque fonction partielle est définie par deux fonctions :

— la fonction `base` détermine le *domaine de définition* qui correspond au sous-ensemble fini sur lequel la fonction est restreinte

— la fonction `find f x` donne l'*image* de `f` pour l'élément `x`.

La fonction `find f x` renvoie `Some y` lorsque la fonction `f` est définie sur l'élément `x` et l'image de `x` par `f` est `y`. Dans le cas où `f` n'est pas définie sur `x`, la valeur renvoyée est `None` Cette propriété est assurée par `find_spec_1`. La fonction `find` renvoie le même résultat si les n-uplets en entrée sont équivalents par l'hypothèse `find_spec_2`.

Chaque variante a aussi une fonction de comparaison `compare` et une fonction `equal` qui vérifie l'égalité entre deux fonctions partielles.

```
  compare : femap → femap → comparison;
  equal : femap → femap → bool;
  compare_spec : ∀ f1 f2, compare f1 f2 = Eq ⟷ equal f1 f2 = true;
  compare_eq_trans : ∀ a1 a2 a3,
    compare a1 a2 = Eq → compare a2 a3 = Eq → compare a1 a3 = Eq;
  compare_eq_lt_trans : ∀ a1 a2 a3,
    compare a1 a2 = Eq → compare a2 a3 = Lt → compare a1 a3 = Lt;
  compare_lt_eq_trans : ∀ a1 a2 a3,
    compare a1 a2 = Lt → compare a2 a3 = Eq → compare a1 a3 = Lt;
  compare_lt_trans : ∀ a1 a2 a3,
    compare a1 a2 = Lt → compare a2 a3 = Lt → compare a1 a3 = Lt;
  compare_lt_gt : ∀ a1 a2, compare a1 a2 = CompOpp (compare a2 a1);
}.
```

La fonction de comparaison satisfait les mêmes propriétés que la fonction de comparaison de l'enregistrement `Oeset.Rcd`. Ce n'est pas un hasard : cela nous permet de définir une relation d'ordre sur les fonctions partielles.

Variante 1

La variante 1 (module `Femap`) contient deux opérations sur les fonctions partielles : `singleton` et `add`.

Singleton

```
  singleton : Elt1 → Elt2 → femap;
  singleton_base :
    ∀ x y,
      base (singleton x y) = Feset.singleton FeElt1 x;
  singleton_find :
    ∀ x y,
      find x (singleton x y) = Some y;
```

`singleton x y` donne la fonction partielle de `Elt1` vers `Elt2` qui est définie sur un seul élément `x` et dont l'image est `y`.

$$\texttt{Elt1} \xrightarrow{\text{singleton x y}} \texttt{Elt2}$$
$$x \longrightarrow y$$

Add

```
add : Elt1 → Elt2 → femap → femap;
add_base : ∀ x y f,
    base (add x y f) = Feset.add FeElt1 x (base f);
add_find_1 : ∀ x x' y f,
    Oeset.eq_bool x x' = false →
    find x' (add x y f) = find x' f;
add_find_2 : ∀ x y f,
    find x (add x y f) = Some y;
```

`add x y f` prend une fonction partielle `f` et renvoie la fonction partielle qui est a le même comportement que `f` sauf pour l'élément `x`. Si `f` n'est pas définie sur l'élément `x`, `add x y f` étend sa définition à `x` et son image est `y`. Si `f` est définie sur l'élément `x`, `add x y f` remplace l'image de `x` par l'élément `y`.

$$\texttt{Elt1} \xrightarrow{\text{add x y f}} \texttt{Elt2}$$
$$x' \longrightarrow y \text{ si x' est équivalent à x}$$
$$x' \longrightarrow f(x') \text{ si x' n'est pas équivalent à x et } x' \in \texttt{base(f)}$$

Variante 2

L'enregistrement de cette variante est nommé `Femap2.Rcd` et prend un argument supplémentaire l'élément `zero`.

A partir de cette variante et pour les variantes suivantes, l'enregistrement comprend la fonction `coeff`. Le but de cette fonction est de pouvoir donner une valeur par défaut (ici `zero`) aux éléments non présents dans le domaine de définition. Cette fonction a la différence de `find` est définie sans type `Option`. `coeff` garde les mêmes images que `find` pour les éléments du domaine de définition .

```
coeff : Elt1 → femap → Elt2;
coeff_spec_1 : ∀ s x y,
    find x f = Some y → coeff x f = y;
coeff_spec_2 : ∀ s x,
    find x f = None → coeff x f = zero;
```

Cette fonction est donc dite presque nulle sauf sur un ensemble fini qui est `base`. On note que `zero` est un élément de `Elt2`. Il peut donc être l'image d'un élément du domaine de définition.

La fonction `coeff` permet aussi d'ajouter une propriété pour l'égalité entre deux fonctions partielles. Ainsi deux fonctions sont égales si et seulement si les extensions de ces deux fonctions coïncident :

```
equal_spec : ∀ f1 f2, equal f1 f2 = true ⟷ (∀ x, eq OE2 (coeff x f1) (coeff x f2));
```

On notera que l'égalité ne prend pas en compte la `base` des deux fonctions. Elles peuvent donc différer entre les deux fonctions. Cela aura un impact sur la définition de certains lemmes du **Chapitre 5**.

Une opération ajoutée à cette variante est la fonction `empty`. Elle renvoie une fonction partielle définie sur aucun élément et dont l'extension donne la fonction constante nulle.

```
empty : femap;
empty_base : base empty = Feset.empty;
```

Variante 3

Cette variante (nommée `Femap3`) propose deux nouvelles fonctions - `union` et `transpo` - et l'enregistrement de cette variante prend deux arguments supplémentaires - `plus` et `mul` de type `Elt2 -> Elt2 -> Elt2`.

Union

Cette variante généralise l'opération `add` avec l'opération `union` qui s'effectue sur deux fonctions.

```
union : femap → femap → femap;
union_base : ∀ f f',
     base (union f f') = Feset.union (base f) (base f');
union_find : ∀ s f' x,
     x ∈ (base (union f f')) →
     find x (union f f') = Some (plus (coeff x f) (coeff x f'));
```

`union f1 f2` est définie sur tous les éléments étant dans la base `f1` ou dans la base de `f2`. L'image d'un élément `x` de la base de union `f1` et `f2` est l'addition des images de `x` par `f1` et `f2` si `x` est dans les bases de `f1` et de `f2`, (`f1 x`) si `x` n'est pas dans la base de `f2` et (`f2 x`) si `x` n'est pas dans la base de `f1`.

$$
\begin{aligned}
\texttt{Elt1} &\xrightarrow{\text{union f1 f2}} \texttt{Elt2} \\
x &\longrightarrow f_1(x) + f_2(x) \text{ si } x \in \texttt{base(f1)} \cap \texttt{base(f2)} \\
x &\longrightarrow f_1(x) + 0 \text{ si } x \in \texttt{base(f1)} \backslash \texttt{base(f2)} \\
x &\longrightarrow 0 + f_2(x) \text{ si } x \in \texttt{base(f2)} \backslash \texttt{base(f1)}
\end{aligned}
$$

Transpo

L'opération `transpo y f` applique la fonction `fun x => mul x y` au résultat de l'application de la fonction `f`.

```
transpo : Elt2 → femap → femap;
transpo_base : ∀ y f,
      base (transpo y f) = base f;
transpo_find : ∀ f x y,
      x ∈ (base (transpo y f)) →
      find x (transpo y f) = Some (mul (coeff x f) y);
```

Pour chaque élément `x` du domaine de définition de `f`, `transpo y f` associe la multiplication de l'image de `x` par `f` avec l'élément `y`.

$$\texttt{Elt1} \xrightarrow{\text{transpo y f}} \texttt{Elt2}$$
$$x \longrightarrow f(x) * y \text{ si } x \in \texttt{base(f)}$$

Cette variante sera utilisée pour formaliser les annotations de la where-provenance dans le **Chapitre 5**.

Variante 4

Cette dernière variante est celle utilisée lors de la formalisation des polynômes **Sous-section 4.3.1**.

Cet enregistrement `Femap4.Rcd` prend deux arguments supplémentaires par rapport à la variante 3. Le premier est une fonction de type `Elt1 -> Elt1 -> Elt1` qu'on note `mul1` pour la suite. Ici la fonction `mul` de la version précédente est noté `mul2`. Le deuxième argument est `OE1_is_CIO : CIO mul1 OE1`.

```
Record CIO (A : Type) (mul : A → A → A) (TA : Oeset.Rcd A) : Prop
  := mk_CIO
  { compat_l : ∀ a1 a2 a3 : A,
                 eq TA a1 a2 ⟷ eq TA (mul a1 a3) (mul a2 a3)}
```

Cet argument assure que la multiplication est compatible avec l'équivalence et si deux multiplications sont équivalentes et l'argument de droite est le même, alors les deux arguments de gauche sont égaux.

Cette variante conserve donc ici les opérations `empty`, `singleton` et `union`. Elle admet une extension de l'opération `transpo` et une nouvelle opération `mul`.

Transpo

L'opération `transpo` a été modifiée. Maintenant, `transpo x y f` renvoie la fonction partielle suivante :

- le domaine de définition de la nouvelle fonction est égal à celui de `f` décalé avec la fonction `fun e => mul1 e x`
- pour chaque élément `x'` de `base f`, l'image de `mul1 x' x` pour la nouvelle fonction est égale à `mul2 (coeff x' f) y`.

```
transpo : Elt1 → Elt2 → femap → femap;
transpo_base : ∀ x y f,
      base (transpo x y f) = Feset.map (fun e ⇒ mul1 e x) (base f);
transpo_find : ∀ f x1 x2 y,
      (mul1 x1 x2) ∈ (base (transpo x2 y f)) →
      find (mul1 x1 x2) (transpo x2 y f) = Some (mul2 (coeff x1 f) y);
```

$$\texttt{Elt1} \xrightarrow{\text{transpo x y f}} \texttt{Elt2}$$

$$x' \longrightarrow f(x'') * y \text{ si } x' \text{ est le résultat de l'opération } \texttt{mul1 x'' x}$$
et `x''` est un élément de `base f`

L'argument `OE1_is_CIO : CIO mul1 OE1` permet d'assurer de l'unicité de l'image d'un élément du domaine de définition pour la fonction renvoyée par `transpo`.
Sans cette hypothèse, nous pouvons avoir deux éléments différents `x1` et `x2` de `base f` tel que `mul1 x1 x3` et `mul1 x2 x3` sont équivalents mais
`mul2 (coeff x1 f) y` et `mul2 (coeff x2 f) y` sont différentes.

Mul

```
mul_Pol : femap → femap → femap;
mul_Pol_spec: ∀ f f',
 mul_Pol f f' = Feset.fold (fun e a ⇒ union (transpo e (coeff e f') f) a) (base f') empty
```

`mul_Pol f f'` applique pour chaque élément e du polynôme `f` l'opération `transpo` e sur l'autre polynôme `f'` et fait l'union des résultats (ce qui reviendra à multiplier un polynôme `f'` par chaque monôme de `f` et additionner le résultat dans la **Section 4.3**.

Instanciation exécutable pour les fonctions partielles

L'instanciation des fonctions partielles présentée dans cette sous-section propose une version exécutable de celles-ci et laisse à l'utilisateur le choix des ensembles de départ `Elt1` et d'arrivée `Elt2` (ainsi que les relations d'ordre et les opérations nécessaires). Ceci revient à laisser abstrait les ensembles de départ et d'arriver et d'instancier les fonctions et les types du corps des enregistrements.

Instanciation commune aux quatre variantes

Le type `femap` est pour chaque variante instanciée par le type `Feset.set (Feset.build OE1) * quotient_function Elt2 OE1`.
`Feset.build` correspond à l'ensemble fini concret du domaine de définition de la fonction partielle et donc à la base d'une fonction.

```
Record quotient_function Elt1 Elt2 OElt1 :=
  mk_qf { f : Elt1 → Elt2;
    equiv_1_eq_2 : ∀ e1 e2, eq OElt1 e1 e2 → f e1 = f e2 }.
```

`quotient_function` contient une fonction totale de `Elt1` vers `Elt2` (qui correspond à `coeff` dans les variantes 2 à 4) et une propriété qui impose que cette fonction soit bien compatible avec l'équivalence de `Elt1`.

L'instanciation de la fonction `find` est une fonction qui vérifie que l'élément est bien dans la base puis donne la valeur si c'est le cas et sinon renvoie `None`.

Je ne détaille pas ici les autres instanciations des fonctions et des propriétés.

3.5 . Conclusion

Ce chapitre a présenté la sémantique de l'algèbre relationnelle positive et de différentes extensions qui seront utilisées dans les chapitres qui suivent. Il a introduit la bibliothèque `Datacert` ainsi que les structures de données et les requêtes qui sont centrales pour être en mesure de formaliser la provenance en Coq. La dernière partie de ce chapitre aborde deux bibliothèques proposant les lemmes impliquant la fonction `fold_left` et les relations d'équivalences et formalisant les fonctions partielles compatibles avec les équivalences.

Le chapitre suivant propose la formalisation d'un premier type de provenance : la How-provenance.

4 - How-provenance

Les formalisations que nous proposons dans cette thèse se placent dans le cadre théorique de l'algèbre relationnelle positive ou une extension de celui-ci pour représenter les données et les opérations sur celles-ci. Ce chapitre est consacré à la première partie de ma contribution, à savoir la formalisation d'un premier type de provenance appelé how-provenance. Ce type de provenance se concentre sur la façon dont les données sont traitées lors d'une requête et utilise des annotations intégrées dans des structures algébriques particulières : les semi-anneaux commutatifs pour l'algèbre relationnelle, et les K-semi-modules (avec K étant un semi-anneau) pour les fonctions d'agrégation. Dans cette section, nous aborderons la sémantique de la how-provenance ainsi que les structures algébriques nécessaires de manière théorique. Par la suite, nous présenterons la formalisation de ces structures et de la how-provenance dans l'assistant de preuves Coq. Cette formalisation sera découpée en deux étapes : une première étape abstraite et généraliste, suivie d'une seconde étape de divers niveaux d'instanciation avec des implémentations plus concrètes et exécutables. Nous prouverons également le théorème d'adéquation, qui établit le lien entre le modèle sans annotations et le modèle avec les annotations plongées dans l'ensemble des entiers naturels. En outre, ce chapitre comportera une démonstration de l'utilisation des bibliothèques instanciées de quelques requêtes sur une base de données. Enfin, nous conclurons cette section par une discussion sur la how-provenance, en comparant son implémentation avec celle de la where-provenance et en évaluant les avantages et les limites de chaque approche.

4.1 . How-provenance

4.1.1 . Tables annotées et how-provenance

Une relation est un multi-ensemble de n-uplets. Elle est usuellement représentée graphiquement sous la forme de tableaux. Cependant, une autre représentation possible d'une relation est une fonction associant à chaque n-uplet le nombre de fois où celui-ci apparaît dans la relation.

Par exemple, la relation r_0 qui stocke les noms de famille et l'âge de personnes vivant dans un certain pays est représentée graphiquement par le tableau 4.1.

Nom	Âge	Pays
Martin	27	France
Durant	81	France
Martin	27	France
Martin	16	Belgique
Chevalier	58	France
Chevalier	58	France

Table 4.1 – relation r_0

Si nous passons à la représentation de la **Table 4.1** sous la forme de fonction, nous obtenons :

$$(Martin, 27, France) \longrightarrow 2$$

$$(Durant, 81, France) \longrightarrow 1$$

$$(Martin, 16, Belgique) \longrightarrow 1$$

$$(Chevalier, 58, France) \longrightarrow 2$$

et 0 pour tous les autres n-uplets.

Cette fonction associe une information particulière à chaque n-uplet : le nombre d'occurrences de chacun d'entre eux. À chaque étape de l'application d'une requête, le nombre d'occurrences d'un n-uplet intermédiaire ou final va dépendre du nombre d'occurrences des n-uplets précédents et de l'opérateur appliqué. L'information "nombre d'occurrences" de chaque n-uplet se propage pas à pas jusque dans la table résultante en étant soumise à des opérations. Il est ainsi possible de suivre l'évolution du nombre d'occurrences de chacun des n-uplets tout au long d'une requête.

Il est donc naturel de vouloir abstraire l'information "nombre d'occurrences" par d'autres types d'informations pour observer les évolutions de celles-ci lors de l'application d'une requête. Cela revient à annoter chaque n-uplet par une information et propager cette annotation selon des règles dépendantes de l'opérateur de l'algèbre et de la nature de l'annotation choisie. Une table annotée est représentée par une fonction de l'ensemble des n-uplets dans l'ensemble des annotations.

Prenons la table suivante :

Nom	Age	Pays
Martin	27	France
Durant	81	France
Martin	16	Belgique
Chevalier	58	France

Table 4.2 – r_1

On peut par exemple l'annoter comme suit :

$$(Martin, 27, France) \longrightarrow Top\ secret$$

$$(Durant, 81, France) \longrightarrow Public$$

$$(Martin, 16, Belgique) \longrightarrow Confidentiel$$

$$(Chevalier, 58, France) \longrightarrow Public$$

La table est annotée par le degré d'accréditation qui est nécessaire pour avoir accès à une donnée.

Ce qui donne graphiquement :

Nom	Age	Pays	Sécurité
Martin	27	France	Top secret
Durant	81	France	Public
Martin	16	Belgique	Confidentiel
Chevalier	58	France	Public

Table 4.3 – r^{annot} : relation r_1 annotée

Si nous appliquons à cette table annotée une requête, par exemple une projection sur la colonne correspondant au nom, la table résultante est annotée comme suit :

$$(Martin) \longrightarrow Confidentiel$$

$$(Durant) \longrightarrow Public$$

$$(Chevalier) \longrightarrow Public$$

Ce qui donne graphiquement :

Nom	Sécurité
Martin	Confidentiel
Durant	Public
Chevalier	Public

Table 4.4 – $\pi_{Pays=\text{“}France\text{”}} r^{annot}$

Nous constatons que les n-uplets $(Martin, 27, France)$ et $(Martin, 16, Belgique)$ ont la même projection sur l'attribut "Nom". Leurs annotations $Top\ secret$ et $Confidentiel$ ont subi une opération pour donner l'annotation $Confidentiel$. Ici cette opération est de choisir le plus faible degré d'accréditation entre les deux annotations. Les autres n-uplets conservent la même annotation qu'avant la projection.

Nous avons maintenant donc des tables annotées par des informations propres à chaque n-uplet et qui sont représentées par des fonctions de l'ensemble des n-uplets dans l'ensemble des annotations. Aussi à chaque étape d'une requête, les annotations subissent des opérations pour maintenir l'information souhaitée.

La notion de tables annotées est intrinsèquement liée à la notion de provenance. Lorsque nous nous intéressons à la provenance d'une donnée résultante de l'application d'une requête, nous cherchons à connaître l'histoire de cette donnée. Par exemple, nous voulons identifier quelles données des tables sources ont eu un impact sur la production de la donnée, comment ces dernières ont été transformées pour obtenir la donnée finale ou encore d'où provient chacune des valeurs de la donnée finale. La how-provenance est un sous-type de provenance qui se concentre sur le "comment" : comment les données sources ont été propagées et comment elles ont été transformées au cours de la requête. La how-provenance cherche ainsi à garder une trace de l'information "comment" pour chaque étape de l'application d'une requête. Cela explique l'utilisation des annotations pour obtenir la how-provenance d'une donnée.

La nature de l'annotation va dépendre de la caractéristique dont l'utilisateur souhaite observer l'évolution au cours des différentes transformation des données. L'utilisateur peut vouloir observer le nombre d'occurrences d'une donnée, son degré de confidentialité ou travailler avec des données incomplètes. Le travail [9] propose de plonger les annotations dans la structure algébrique des semi-anneaux commutatifs et présente une sémantique de la how-provenance détaillant les opérations appliquées aux annotations pour chaque opérateur de l'algèbre relationnelle positive. Ce travail est complété par [10] qui propose d'étendre l'expressivité des opérateurs de l'algèbre avec les fonctions d'agrégation ce qui nécessite de passer d'une structure de semi-anneau à celle de semi-module. Le semi-anneau ou semi-module choisi correspond à la caractéristique des données que l'utilisateur souhaite observer.

Avant de détailler le contenu de ces deux travaux, nous proposons dans la sous-section qui suit d'observer plus en détail comment la propagation des annotations s'effectue lorsque l'information est le nombre d'occurrences d'un n-uplet pour une requête donnée. Cela permet d'avoir l'intuition de la structure algébrique utilisée pour représenter les annotations dans le cadre de la how-provenance ainsi que les différentes opérations appliquées aux annotations pour conserver une information lors de l'application d'une requête.

4.1.2 . Propagation des annotations

Notons tout d'abord que le nombre d'occurrences d'un n-uplet peut seulement prendre des valeurs entières positives. Les annotations dans ce cas sont donc plongées dans l'ensemble des entiers naturels (que je note $\mathbb{N}$ dans la suite) qui est un semi-anneau commutatif. Les tables annotées par le nombre d'occurrences d'un n-uplet sont donc des fonctions de l'ensemble des n-uplets vers l'ensemble $\mathbb{N}$.

Remarque

Nous observons que le nombre 0 a un rôle particulier dans le cas du nombre d'occurrences des n-uplets. Lorsqu'un n-uplet n'apparaît pas dans une table, son nombre d'occurrences est égal à zéro dans cette table. Et réciproquement, si un n-uplet apparaît dans une table, son nombre d'occurrences est forcément strictement positif. Une annotation nulle est donc le critère informant de l'absence d'un n-uplet dans une table.

How-provenance

Lorsqu'on se place dans le cadre de la how-provenance pour généraliser l'intuition, l'ensemble des annotations représentant l'information "comment" est noté K. Dans [14], les auteurs font la distinction entre les données étant présentes et les données absentes d'une relation à l'aide d'une annotation particulière : cet élément particulier de K est noté dans la suite 0_K.

Attaquons-nous maintenant à la propagation des annotations dans le cadre de l'algèbre relationnelle positive puis nous élargirons le cadre avec les extensions présentées dans le chapitre précédent.

Tables sources

Nous présentons ici les tables sources de notre exemple.

	Patient	**Maladie**	**Ann**
t_1	Pat.1	Rhume	a_1
t_2	Pat.51	Varicelle	a_2
t_3	Pat.59	Otite	a_3
t_4	Pat.1	Otite	a_4

Table 4.5 – relation r_3

	Patient	**Maladie**	**Ann**
t_5	Pat.67	Angine	a_5
t_6	Pat.1	Otite	a_6
t_7	Pat.314	Gastro	a_7

Table 4.6 – relation r_4

	Patient	**Médic**	**Ann**
t_8	Pat.1	Goutte	a_8
t_9	Pat.59	AB	a_9
t_{10}	Pat.314	Goutte	a_{10}
t_{11}	Pat.67	Doliprane	a_{11}

Table 4.7 – relation r_5

Chaque table source a pour chaque ligne une annotation a_i où i est le numéro du n-uplet source correspondant. Dans le cadre des nombres d'occurrences, les a_i sont des nombres entiers prédéfinis dans la base de données.

How-provenance

Pour la how-provenance, les annotations sont dans l'ensemble K.

Opérateur du n-uplet vide $\epsilon_{<>}$

	Annot
t_{12}	1

Table 4.8 – $\epsilon_{<>}$

La table associée à l'opérateur ϵ est toujours la même et ne contient qu'un seul n-uplet. Le nombre d'occurrences pour le n-uplet vide est donc de 1 et pour les autres n-uplets est de 0. L'annotation pour le n-uplet vide dans cette table ne peut prendre qu'une seule valeur.

How-provenance

L'opérateur ϵ donnant toujours le même résultat, l'annotation du n-uplet vide est toujours la même et constitue le second élément particulier de K que l'on note 1_K et qui est différent de 0_K (car le n-uplet vide apparaît dans la table).

La table annotée correspondant à la table résultante de cet opérateur est représentée par la fonction :

$$t \mapsto \begin{cases} 1_K & \text{si } t =<> \\ 0_K & \text{sinon} \end{cases}$$

Le renommage ρ_r.

	Patient	**Organe**	**Annot**
t'_1	Pat.1	Rhume	a_1
t'_2	Pat.51	Varicelle	a_2
t'_3	Pat.59	Otite	a_3
t'_4	Pat.1	Otite	a_4

Table 4.9 – renommage $\rho_{[Maladie \to Organe]}(r_3)$

Pour l'opérateur de renommage, le nombre d'occurrences associé à un n-uplet n'est pas modifiée entre la table de la sous-requête et la table résultante. Seule la sorte du n-uplet a changé. Si le n-uplet t_1 apparaît 5 fois dans la **Table 4.5** alors t'_1 apparaît 5 fois après le renommage.

How-provenance

Du point de vue de la how-provenance, aucune nouvelle donnée n'est produite, modifiée ou supprimée. Seuls les noms des colonnes changent. L'opérateur de renommage n'impactant pas les valeurs des données, les annotations des données restent les mêmes.

Je définis la fonction de renommage inverse r^{-1}, pour une fonction de renommage r de la façon suivante :

— $\varnothing^{-1} = \varnothing$

— $(a := b, r)^{-1} = b := a, r^{-1}$ où a et b sont des attributs et où r est une fonction de renommage

La fonction pour le renommage $\rho_r(q)$ est donc :

$$t \mapsto \textbf{annot}(q, t[r^{-1}])$$

où **annot**(q,t) est l'annotation du n-uplet t dans la table associée à la sous-requête q.

La projection $\pi_A(\cdot)$

	Maladie	**Annot**
t_{13}	Rhume	a_1
t_{14}	Varicelle	a_2
t_{15}	Otite	$a_3 + a_4$

Table 4.10 – projection $\pi_{Maladie}(r_3)$

Lors d'une projection, le n-uplet résultant t dépend des n-uplets initiaux dont la projection est égale à t. Ainsi le nombre de fois où apparaît t dans la table résultante est égal à la somme des occurrences de ces n-uplets initiaux. Dans la **Table 4.5**, nous observons que la projection de t_3 et t_4 sur l'attribut "Maladie" donne le n-uplet t_{15} de la **Table 4.10**. Si t_3 apparaît 3 fois dans la **Table 4.5** et t_4 apparaît 4 fois alors dans la **Table 4.10**, t_{15} apparaît 7 fois. Il y a donc une opération qui s'applique sur le nombre d'occurrences des deux n-uplets ici qui est l'addition. De plus, si t_4 n'apparaît pas dans la **Table 4.5**, le nombre d'occurrences de t_{15} est le même que celui de t_3 - 0 étant neutre pour l'opération $+$.
Si nous nous attardons sur la production de la donnée t_{15}, c'est qu'elle dépend des deux n-uplets t_3 et t_4 et la présence d'un des deux n-uplets suffit à ce que cette donnée soit présente dans la table résultante.

How-provenance

Pour refléter ces deux observations dans la sémantique de la how-provenance, [9] utilise une opération $+_K$ qui est appliquée sur les annotations des n-uplets dont la projection est égale au n-uplet d'intérêt et impose que l'élément 0_K est neutre pour l'opérateur $+_K$.
La fonction associée à une table annotée après une projection $\pi_A(q)$ est :

$$t \mapsto \sum_{t' | t = t'_{|A}} \textbf{annot}(q, t')$$

La sélection $\sigma_f(\cdot)$

	Patient	**Maladie**	**Annot**
t_3	Pat.59	Otite	a_3
t_4	Pat.1	Otite	a_4

Table 4.11 – sélection $\sigma_{Maladie=Otite}(r_3)$

Pour l'opérateur de sélection, un n-uplet qui apparaît dans la table résultante - donc qui vérifie le prédicat - a le même nombre d'occurrences que ce même n-uplet dans la table liée à la sous-requête. Les n-uplets qui ne vérifient pas le prédicat ont un nombre d'occurrences nul.

How-provenance

L'annotation des n-uplets ne vérifiant pas le prédicat et donc n'étant pas dans la table résultante, est 0_K. L'application de l'opérateur de sélection ne produit pas de nouvelles données et ne modifie pas les anciennes. Les annotations des données présentes dans la table résultante ne subissent pas de modification. La fonction associée à une table annotée après une sélection $\sigma_P(q)$ est :

$$t \mapsto \begin{cases} \textbf{annot}(q, t) & \text{si } P(t) = V \\ 0_K & \text{sinon} \end{cases}$$

La jointure naturelle $\cdot \bowtie \cdot$

Un n-uplet résultant de la jointure naturelle de deux tables est une fusion entre deux des n-uplets des deux tables. On peut déjà en déduire qu'il y a une opération entre les deux annotations pour la how-provenance.
Du point de vue du nombre d'occurrences, on effectue une multiplication entre les nombres d'occurrences. Cette opération est différente de celle utilisée pour la projection.

	Patient	Maladie	Médic	Annot
t_{17}	Pat.1	Rhume	Goutte	$a_1 \times a_8$
t_{18}	Pat.59	Otite	AB	$a_3 \times a_9$
t_{19}	Pat.1	Otite	Goutte	$a_4 \times a_8$

Table 4.12 – jointure naturelle $r_3 \bowtie r_5$

How-provenance

Pour produire un n-uplet t dans le cas d'une jointure naturelle $q_1 \bowtie q_2$, il est nécessaire que les projections $t_{|\mathbf{Attr_q}(q_1)}$ et $t_{|\mathbf{Attr_q}(q_2)}$ soient présentes dans les tables associées à q_1 et q_2 respectivement. Cela diffère de la production d'un n-uplet t résultant d'une projection : la présence de t dans la table résultante était assurée si au moins un n-uplet dont la projection était égale à t, était dans la table initiale. Nous avons donc une nouvelle opération notée $\times_K$ qui s'applique sur les annotations des n-uplets liés aux sous-requêtes pour la how-provenance. La fonction associée à $q_1 \bowtie q_2$ est :

$$t \mapsto \mathbf{annot}(q_1, t_{|\mathbf{Attr_q}(q_1)}) \times_K \mathbf{annot}(q_2, t_{|\mathbf{Attr_q}(q_2)})$$

On notera que l'opération $+_K$ est utilisée dans le cas où un seul des n-uplets initiaux est nécessaire pour produire le n-uplet résultant et l'opération $\times_K$ si plusieurs n-uplets doivent être simultanément présents dans les tables initiales.

L'union $\cdot \cup \cdot$

	Patient	Maladie	Annot
t_1	Pat.1	Rhume	a_1
t_2	Pat.51	Varicelle	a_2
t_3	Pat.59	Otite	a_3
$t_{4,6}$	Pat.1	Otite	$a_4 + a_6$
t_5	Pat.67	Angine	a_5
t_7	Pat.314	Gastro	a_6

Table 4.13 – union $r_3 \cup r_4$

Le nombre de fois où le n-uplet apparaît dans la table résultante est l'addition du nombre d'occurrences de ce n-uplet dans les tables des sous-requêtes. On retrouve l'opération d'addition comme pour la projection.

How-provenance

Du point de vue de la how-provenance, les n-uplets ne sont pas modifiés mais peuvent provenir d'une ou des deux tables des sous-requêtes. L'opération effectuée ici est la même que celle que pour la projection, le $+_K$. La fonction associée à $q_1 \cup q_2$ est donc :

$$t \mapsto \textbf{annot}(q_1, t) +_K \textbf{annot}(q_2, t)$$

La structure de l'ensemble K des annotations pour la how-provenance est constituée d'au moins deux éléments distincts 0_K et 1_K et a deux opérations $+_K$ et $\times_K$. On a déjà constaté que l'élément 0_K est neutre pour $+_K$. Le caractère de semi-anneau est dû aux équivalences algébriques entre les opérateurs de l'algèbre. En effet, les requêtes étant algébriquement équivalentes, les données des tables qui leur sont associées sont les mêmes. Les auteurs de [9], ont souhaité conserver cette propriété sur les annotations qui ne varient donc pas à équivalence algébrique près, ce qui donne les caractères associatif, commutatif, distributif, neutre et absorbant des opérations et des éléments de K. Pour plus de détails sur la structure de semi-anneau, voir la **Section 4.2**. La structure des semi-anneaux commutatifs a émergé pour refléter dans les annotations les propriétés d'équivalences algébriques. Un n-uplet final a donc la même annotation si les requêtes, du résultat desquelles il fait partie, sont équivalentes.

Remarque

Comme précisé dans le **Chapitre 3**, notre cadre est l'algèbre relationnelle positive et ne contient pas la différence relationnelle. J'ai choisi de ne pas étendre l'algèbre relationnelle à la différence car la sémantique de la how-provenance ne peut être étendue avec l'opérateur de la différence pour donner un cadre instanciable par tous les semi-anneaux couramment utilisés et qui vérifient les propriétés d'équivalence algébrique [31].

Extension aux expressions d'attributs

Lors de l'extension des opérateurs de l'algèbre relationnelle positive aux expressions arithmétiques sur des attributs, la propagation des annotations de how-provenance n'est modifiée que pour l'opérateur η. Le raisonnement pour le cas de la sélection reste le même.

Les n-uplets de la table résultante de l'application de l'opérateur η sont produits de manière similaire à la production des n-uplets lors de l'application de l'opérateur de projection dans l'algèbre relationnelle. L'opération choisie dans ce cas est $+_K$ et s'effectue sur toutes les annotations des n-uplets dont le renommage est égal à la donnée d'intérêt.

La fonction associée à $\eta[r]$ est donc :

$$t \mapsto \sum_{t'|t=t'[r]} \textbf{annot}(q, t')$$

Extenstion aux fonctions d'agrégation

Dans cette thèse, nous abordons seulement les requêtes pour lesquelles les fonctions d'agrégation sont forcément en tête de la requête, c'est-à-dire qu'aucun opérateur d'une requête ne peut avoir en sous-requête une fonction d'agrégation. Nous ne capturons donc pas toute la sémantique de la how-provenance pour SQL. Ainsi les requêtes $\sigma_P(r_1 \cup (r_2 \bowtie r_3))$ et $\Gamma_{r,gb}(r_1 \cup \epsilon_{<>})$ sont dans notre restriction mais pas $\sigma_P(\Gamma_{r,gb}(r_1))$.

Contrairement aux autres opérateurs, les annotations choisies dans le cas de l'application d'une fonction d'agrégation ont un impact sur les valeurs des n-uplets résultants.

	A	**Annot**
t_1	18	3
t_2	37	2
t_3	5	4

Table 4.14 – relation r_6

	A	**Annot**
t'_1	18	1
t'_2	37	1
t'_3	5	1

Table 4.15 – relation r_7

Somme	**Annot**
148 = $18 \times 3 + 37 \times 2 + 5 \times 4$	1

Table 4.16 – Somme$_A(r_6)$

Somme	**Annot**
60 = $18 \times 1 + 37 \times 1 + 5 \times 1$	1

Table 4.17 – Somme$_A(r_7)$

Pour notre exemple, les annotations correspondent aux nombres d'occurrences d'un n-uplet. Les **Tables 4.14 et 4.15** ne diffèrent que par leurs annotations. Cependant, les tables résultantes (**Table 4.16** et **Table 4.17**) de l'application de la somme de la colonne A donnent deux n-uplets différents car nous faisons la somme des valeurs de tous les n-uplets. La multiplicité d'un n-uplet (et donc son annotation) est prise en compte lors du calcul de la somme.

Lors de l'application de fonctions d'agrégation, les annotations et les valeurs finales dépendent des annotations initiales. Nous observons aussi que le calcul des valeurs se fait avec deux opérations $+$ et $\times$.

Autre exemple, la **Table 4.18** est annotée par des entiers naturels $a_1, .., a_4$. La table résultante **Table 4.19** de l'application de la fonction "Max" est différente selon l'annotation des n-uplets sources. Si tous les n-uplets sont présents dans la table source, les n-uplets obtenus sont (29, Vrai) et (37,Faux) avec multiplicité 1.

	A	B	Annot
t_1	18	Vrai	a_1
t_2	37	Faux	a_2
t_3	5	Faux	a_3
t_4	29	Vrai	a_4

Table 4.18 – relation annotée r_8

	Max	B	Annot
t_5	$a_1 \otimes 18 \oplus a_4 \otimes 29$	Vrai	$\delta(a_1 + a_4)$
t_6	$a_2 \otimes 37 \oplus a_3 \otimes 5$	Faux	$\delta(a_2 + a_3)$

Table 4.19 – [Maximum$_A(r_8), B$] groupé sur B renommé "Max"

Si le n-uplet t_4 n'est pas dans la table initiale, alors les n-uplets obtenus sont (18, Vrai) et (37,Faux) avec multiplicité 1. La valeur de t_5 change mais pas l'annotation. Nous observons que les valeurs sont encore calculées à partir de deux opérations : $\oplus$ qui correspond au maximum et $\otimes$ qui vaut zéro si la première opérande est nulle et vaut la deuxième opérande sinon. Ces deux opérations sont différentes de celles utilisées pour la fonction "Somme".

Si les n-uplets t_1 et t_4 ne sont pas dans la table initiale, alors le n-uplet résultant est (37,Faux) avec multiplicité 1. L'annotation est ici passée à 0.

Nous avons donc une fonction δ pour les annotations qui prend en compte toutes les annotations des n-uplets initiaux ayant un impact sur le n-uplet final. De plus, si toutes les annotations des n-uplets contributeurs sont nulles, alors la fonction δ donne 0. Pour prendre en compte les n-uplets contributeurs, les différentes annotations sont sommées avant d'être prises en argument par la fonction δ.

How-provenance

Le travail [10] suggère que les valeurs d'une colonne issues de l'application d'une fonction d'agrégation sont plongées dans un K-semi-module si les annotations sont plongées dans le semi-anneau K. Ce semi-module a donc deux opérateurs $\oplus_K$ et $\otimes_K$. Le semi-module dépend de la fonction d'agrégation et de l'ensemble K. Comme pour la structure de semi-anneau, la structure de semi-module provient de la volonté des auteurs de conserver les mêmes annotations entre deux requêtes équivalentes.

Les valeurs résultantes de l'application d'une fonction d'agrégation $agg(a)$ sont donc de la forme : $a_1 \otimes v_1 \oplus .. \oplus a_n \otimes v_n$ où a_i sont les annotations des n-uplets contribuant à la création du n-uplet et v_i les valeurs de ces n-uplets pour l'attribut a. Pour ce qui est de la valeur d'un n-uplet pour une expression d'attributs faisant partie du groupement gb, elle est égale à la valeur commune du groupe pour cet attribut.

Pour ce qui est des annotations, les n-uplets d'un groupement contribuant à un n-uplet ont leur annotations additionnées, puis le résultat est donné en argument à la fonction δ_K. L'annotation totale d'un n-uplet résultant correspond à la somme des annotations associées aux groupement contribuants. La fonction δ_K est dépendante du semi-anneau choisi. On remarque que lorsque toutes les annotations initiales sont nulles (c'est-à-dire que les n-uplets n'apparaissent pas) alors le n-uplet produit n'apparaît pas dans le résultat. Son annotation est donc 0_K. Ce qui nous permet d'en déduire que $\delta(0_K) = 0_K$. Dans le cas particulier des annotations dans les entiers naturels, la fonction δ^N renvoie 0 si son argument est 0 et 1 sinon.

4.1.3 . Sémantique de la how-provenance

Dans cette sous-section, nous présentons la sémantique de la how-provenance. Les fonctions des n-uplets vers les annotations sont nommées K-relations.

Définition 4.1: K-relations

Soit $(K, 0, 1, +, \times)$ un semi-anneau commutatif. Une K-relation est une fonction quasi-nulle de l'ensemble des n-uplets vers l'ensemble K.

Nous notons K-Algèbre, l'algèbre relationnelle positive avec les annotations plongées dans un semi-anneau commutatif K.

Définition 4.2: Sémantique (K-algèbre)

On suppose les requêtes bien formées. Soit $(K, 0, 1, +, \times)$ un semi-anneau commutatif. Soient q, q_1 et q_2 des requêtes, f une formule. Chaque fonction est une K-relation.

- $[\![r]\!]^K_{hw} = t \mapsto$ annot (r, t)
  où $annot$ est la fonction qui associe une annotation à chaque n-uplet t d'une relation source r
- $[\![\epsilon_{<>}]\!]^K_{hw} = t \mapsto \begin{cases} 1 & \text{si } t = <> \\ 0 & \text{sinon} \end{cases}$
- $[\![\rho_r(q)]\!]^K_{hw} = t \mapsto [\![q]\!]^K_{hw}(t[r^{-1}])$
  où r est une fonction de renommage
- $[\![\pi_A(q)]\!]^K_{hw} = t \mapsto \sum_{t' | t = t_{|A}} [\![q]\!]^K_{hw}(t')$
  où A est un ensemble d'attributs
- $[\![\sigma_f(q)]\!]^K_{hw} = t \mapsto \begin{cases} [\![q]\!]^K_{hw}(t) & \text{si } eval(f,t) = \top \\ 0 & \text{sinon} \end{cases}$
  où $eval$ évalue la formule logique f sur le n-uplet t
- $[\![q_1 \bowtie q_2]\!]^K_{hw} = t \mapsto [\![q_1]\!]^K_{hw}(t_{|\mathbf{Attr_q}(q_1)}) \times [\![q_2]\!]^K_{hw}(t_{|\mathbf{Attr_q}(q_2)})$
- $[\![q_1 \cup q_2]\!]^K_{hw} = t \mapsto [\![q_1]\!]^K_{hw}(t) + [\![q_2]\!]^K_{hw}(t)$

Pour définir la sémantique pour l'opérateur Γ, nous avons besoin de déterminer les groupes contribuant à la production d'un n-uplet résultant. Pour cela, il nous

faut définir dans un premier temps comment s'effectue un groupement lorsqu'on a des K-relations au lieu des relations. Chaque groupe de n-uplets doit avoir les mêmes valeurs pour les expressions de gb et ne retenir que les n-uplets présents dans la table, donc les n-uplets ayant une annotation non nulle. De plus, nous retenons seulement les groupements ayant au moins un n-uplet.

Définition 4.3: Groupement selon gb pour la how-provenance

Soit q une requête et gb un ensemble fini d'expressions d'attributs.
On suppose la requête bien formée et que les attributs contenus dans gb sont dans la sorte de q.

$$Group^{hw}(q, gb) = \{r | \exists (v_1, .., v_n) \in val^{expr}(gb), \forall t, t \in r \Leftrightarrow ([t_{|gb}]_e = (v_1, ..., v_n) \wedge [\![q]\!]^K_{hw}(t) \neq 0_K) \wedge r \neq \varnothing\}$$

où val^{expr} est la fonction qui associe à un ensemble fini d'expressions d'attributs les ensembles des valeurs correspondantes.

Dans un deuxième temps, nous devons calculer les n-uplets obtenus à partir d'un groupement. Pour cela, nous définissons la fonction Ksm qui à une fonction d'agrégation f associe le K-semi-module adéquat. Le calcul des valeurs du n-uplet résultant de l'évaluation d'un groupement est présenté ci-après.

Définition 4.4: Evaluation des valeurs

Soit q une requête et r un ensemble de n-uplets. Soit $a_1, .., a_n$ sont des attributs. Soit fe, fe_1,.., fe_n des fonctions d'agrégation ou des expressions d'attributs.
On suppose que r contient au moins un n-uplet qu'on nomme t^i. On note $Agg^{hw}(r, q, fe)$ l'évaluation de fe sur l'ensemble des n-uplets r selon les annotations donnée par q.

- si fe est la fonction d'agrégation $agg(a)$ alors :
  $Agg^{hw}(r, q, fe) = \sum^{\oplus_W}_{t \in r} [t_{|a}]_e \otimes_W [\![q]\!]^K_{hw}(t)$
  où $Ksm(fe) = (W, 0_W, \oplus_W, \otimes_W)$
- si fe est une expression d'attributs alors :
  $Agg^{hw}(r, q, fe) = [t^i_{|fe}]_e$

L'évaluation de plusieurs fonctions d'agrégation ou d'expression d'attributs sur l'ensemble des n-uplets r selon les annotations données par q est la suivante :

$$Agg^{hw}_t(r, q, (fe_1 := a_1, .., fe_n := a_n)) = (a_1 \mapsto Agg^{hw}(r, q, fe_1), ..., a_n \mapsto Agg^{hw}(r, q, fe_n))$$

Avec les définitions précédentes, on peut définir l'ensemble des groupes contribuant à la production d'un n-uplet résultant.

Définition 4.5: Ensembles des groupes contributeurs

Soit t un n-uplet, q une requête et gb un ensemble fini d'expressions d'attributs. $fe_1,..,\ fe_n$ sont soit des fonctions d'agrégation soit des expressions d'attributs. $a_1,..,a_n$ sont des attributs.
L'ensemble des groupes de la requête q selon le groupement gb contribuant à l'obtention de t est défini par :

$$Cont(t,q,gb,(fe_1 := a_1,..,fe_n := a_n)) = \{r \in Group^{hw}(q,gb) | t = Agg_t^{hw}(r,q,(fe_1 := a_1,..,fe_n := a_n)\}$$

Avec les extensions des opérateurs aux expressions d'attributs et aux fonctions d'agrégation, la sémantique est modifiée pour les opérateurs η^{expr} et Γ de la façon suivante :

Définition 4.6: Sémantique étendue

On suppose les requêtes bien formées. Soit q une requête, r une fonction de renommage, FE des fonctions d'agrégations ou des expressions d'attributs avec renommage et gb un ensemble fini d'expressions d'attributs.

— $[\![\eta_r^{expr}(q)]\!]_{hw}^K = t \mapsto \sum_{t'|t=t'[r]} [\![q]\!]_{hw}^K(t')$

— $[\![\Gamma_{FE,gb}(q)]\!]_{hw}^K = t \mapsto \sum_{r|r\in Cont(t,q,gb,FE)} \delta^K(\sum_{t\in r} [\![q]\!]_{hw}^K(t))$

4.2 . Semi-anneaux commutatifs

4.2.1 . Structure algébrique

Nous rappelons ici les définitions des structures algébriques utilisées dans cette partie.

Définition 4.1: Monoïde $(M,e,*)$

Un monoïde $(M,e,*)$ est un ensemble M muni d'une loi de composition interne $*$ et d'un élément particulier e vérifiant :

— e est neutre pour l'opération $*$: $\forall m \in M,\ m*e = e*m = m$

— $*$ est associative : $\forall m_1,m_2,m_3 \in M^3,\ m_1*(m_2*m_3) = (m_1*m_2)*m_3$

Définition 4.2: Commutativité

Une loi de composition interne $*$ est commutative si et seulement si :
$\forall m_1,\ m_2 \in M^2,\ m_1 * m_2 = m_2 * m_1$

La définition d'un monoïde commutatif est simplifiable pour l'élément neutre à $\forall m,\ m * e = m$ grâce au caractère commutatif.

Définition 4.3: Semi-anneau $(A, 0, 1, +, \times)$

Un semi-anneau $(A, 0, 1, +, \times)$ est un ensemble A muni de deux lois de composition internes $+$ et $\times$ et de deux éléments particuliers 0 et 1 vérifiant :
- $(A, 0, +)$ est un monoïde commutatif
- $(A, 1, \times)$ est un monoïde
- 0 est absorbant pour l'opération $\times$: $\forall m \in M,\ m \times 0 = 0 \times m = 0$
- $\times$ est distributif sur $+$:
  $\forall m_1, m_2, m_3 \in M^3,\ m_1 \times (m_2 + m_3) = (m_1 \times m_2) + (m_1 \times m_3)$
  et $\forall m_1, m_2, m_3 \in M^3,\ (m_1 + m_2) \times m_3 = (m_1 \times m_3) + (m_2 \times m_3)$

Un semi-anneau devient commutatif si la loi interne $\times$ est commutative. De même que pour le monoïde, le caractère commutatif du semi-anneau permet de simplifier sa définition pour l'élément neutre et le caractère distributif.

Affinement du cadre théorique

Dans la **Sous-section 3.4.2**, nous avons évoqué le fait qu'un n-uplet annoté avec l'élément 0 correspond à l'absence de ce n-uplet dans une table [9]. Nous devons donc avoir cette propriété qui est maintenue quelle que soit la requête. Or le cadre théorique en état laisse échapper trois cas qui empêchent de s'en assurer.

J'ai donc ajouté deux propriétés supplémentaires que les semi-anneaux (respectivement semi-modules) doivent vérifier pour pallier ce problème. Une autre propriété sera ajoutée pour la fonction δ. Prenons l'union de deux tables r_9

A	Ann
t	a

Table 4.20 – relation r_9

A	Ann
t	b

Table 4.21 – relation r_{10}

A	Ann
t	$a + b$

Table 4.22 – $r_9 \cup r_{10}$

et r_{10}. On suppose que le n-uplet t est dans au moins l'une des tables sources, donc le n-uplet t est présent dans l'union des deux tables. Comme une annotation

nulle signifie que le n-uplet n'est pas dans la table, l'annotation $a + b$ doit être non nulle. Autrement dit, si l'addition de deux annotations est nulle, alors les deux annotations sont égales à zéro. La première propriété que la structure algébrique doit vérifier est la suivante :

Propriété 4.1: Intégrité de l'addition

$\forall x, y \in R^2$, si $x + y = 0$ alors $x = 0$ et $y = 0$

De même, la jointure bien formée de deux n-uplets compatibles et présents dans les deux tables sources apparaît dans la table résultante donc l'annotation de ce n-uplet final doit être non nulle. On le voit bien dans l'exemple suivant : si le n-uplet t_1 et le n-uplet t_2 sont présents respectivement dans les tables r_{11} et r_{12} alors t_3 est présent dans la table résultante de la jointure naturelle entre r_{11} et r_{12}.

	A	B	Ann
t_1	t_A	t_B	a

Table 4.23 – relation r_{11}

	B	C	Ann
t_2	t_B	t_C	b

Table 4.24 – relation r_{12}

	A	B	C	Ann
t_3	t_A	t_B	t_C	$a \times b$

Table 4.25 – $r_{11} \cup r_{12}$

Autrement dit, si le n-uplet t_3 n'est pas présent dans la table $r_{11} \cup r_{12}$ alors au moins un des n-uplets, t_1 ou t_2, n'est pas présent dans respectivement r_{11} ou r_{12}. La structure algébrique doit donc vérifier la deuxième propriété suivante :

Propriété 4.2: Intègrité de la multiplication

$\forall x, y \in R^2$, si $x \times y = 0$ alors $x = 0$ ou $y = 0$

Avec des arguments similaires, nous constatons que la fonction δ doit aussi vérifier que :

Propriété 4.3: Intègrité de δ

$\forall x \in R$, si $\delta(x) = 0$ alors $x = 0$

4.2.2 . Formalisation abstraite en Coq

Comme ce travail est basé sur la bibliothèque Datacert, les structures algébriques sont compatibles avec la bibliothèque Coccinelle qui formalise les objets

mathématiques utilisés dans Datacert. Les structures de cette bibliothèque ont chacune une relation d'équivalence totale et se trouvent sous la forme d'enregistrement. Je propose ainsi la bibliothèque `SemiRing` qui formalise des semi-anneaux commutatifs en Coq qui préserve l'essence de la bibliothèque Coccinelle. Le premier élément formalisé est une structure de base des semi-anneaux commutatifs : le monoïde commutatif.

```
Record CM (M : Type) (zero : M) (plus : M → M → M) (OM : Oeset.Rcd M) :=
  mk_CM
    {
      plus_assoc : ∀ a1 a2 a3, eq (plus a1 (plus a2 a3)) (plus (plus a1 a2) a3);
      plus_0_l : ∀ a, eq (plus zero a) a;
      plus_comm : ∀ a1 a2, eq (plus a1 a2) (plus a2 a1);
      plus_compat : ∀ a1 a2 b1 b2, eq a1 a2→  eq b1 b2 →
                                      eq (plus a1 b1) (plus a2 b2)
    }.
```

Comme le propose la définition d'un monoïde commutatif, on retrouve les éléments propres au monoïde : l'ensemble `M`, l'élément neutre `zero` et l'opération `plus`. Les trois premiers champs correspondent à la définition du monoïde. Le dernier champ est spécifique de l'utilisation d'une relation d'équivalence `OM` : il faut donc ajouter les cas qui sont masqués lorsqu'on prend une égalité syntaxique habituelle. `plus_compat` impose la compatibilité de l'égalité de `OM` avec le `plus`.

La structure de semi-anneau commutatif est donc formalisée en Coq, comme ci-après :

```
Record CSR (R : Type) (zero one : R) (plus mul : R → R → R) (OR : Oeset.Rcd R) :=
  mk_SC
    {
      diff_0_1 : not (eq zero one);
      isCM_plus : CM zero plus OR;
      isCM_mul : CM one mul OR;
      mul_distr_plus_l : ∀ a1 a2 b,
              eq (mul b (plus a1 a2)) (plus (mul b a1) (mul b a2));
      mul_0_r : ∀ a, eq (mul a zero) zero
    }.
```

On a donc ici un élément `one` et une opération `mul` supplémentaires. Le champ 1 impose des éléments `zero` et `one` distincts. `(R,zero,plus,OR)` et `(R,one,mul,OR)` sont des monoïdes commutatifs par les champs 2 et 3. Les deux derniers champs ajoutent les deux autres propriétés nécessaires pour obtenir un semi-anneau commutatif.

Cette bibliothèque contient aussi quelques lemmes que je n'énonce pas ici pour faciliter la manipulation des éléments des monoïdes ou des semi-anneaux.

Elle contient aussi les preuves que les ensembles usuels de la littérature utilisés comme annotations sont bien des semi-anneaux commutatifs.

Pour ce qui est du caractère intègre de l'addition et de la multiplication, j'utilise un autre enregistrement que chaque semi-anneau devra vérifier.

```
Record Integ (A : Type) (zero : A) (plus : A → A → A) (mul : A → A → A)
    (OA : Oeset.Rcd A) :=
mk_integ
  {
    plus_integ : ∀ a1 a2,
       eq (plus a1 a2) zero = Eq → eq a1 zero = Eq ⋀ eq a2 zero = Eq;
    mul_integ :
       ∀ a1 a2, eq (mul a1 a2) zero = Eq → eq a1 zero = Eq ⋁ eq a2 zero = Eq
  }.
```

4.2.3 . Instances de semi-anneaux commutatifs

Comme vu précédemment, la caractéristique que l'utilisateur cherche à observer impose le semi-anneau choisi pour les annotations. On retrouve les entiers naturels $(\mathbb{N}, 0, 1, +, \times)$ lorsque nous voulons annoter une base de données par le nombre d'occurrences d'un n-uplet et donc par extension permet de représenter la sémantique multi-ensembliste. A contrario, les booléens $(\mathbb{B} = \{\top, \bot\}, \bot, \top, \vee, \wedge)$ informe de la présence ou de l'absence d'un n-uplet et représentent la sémantique ensembliste. L'ajout d'une valeur d'incertitude aux booléens classiques $(\mathbb{B}^3 = \{\top, \bot, ?\}, \bot, \top, \vee^3, \wedge^3)$ permet de représenter des tables avec des données dont la présence est incertaine et donc de représenter les tables incomplètes. L'utilisateur peut aussi vouloir indiquer le degré d'accréditation $(\mathbb{S} = 0_S, 1_S, min_S, max_S)$ requis pour l'accès à certaines données avec le semi-anneau de sécurité ou encore calculer le coût de production d'une donnée avec les entiers tropicaux $(\mathbb{N}^\infty, \infty, 0, min^\infty, +^\infty)$.

Le semi-anneau des polynômes à coefficients entiers $(\mathbb{P}[X], 0, 1, +, \times)$ constitue un cadre général pour tous les semi-anneaux.

A	B	Ann
1	2	a || 4

Table 4.26 – relation r_{13}

A	B	Ann
1	2	b || 3
3	4	c || 6

Table 4.27 – relation r_{14}

A	B	Ann
1	2	$a + b$ || 7
3	4	c || 6

Table 4.28 – $r_{13} \cup r_{14}$

Les n-uplets des **Tables de 4.26 à 4.28** sont alternativement annotés par une indéterminée (et donc plongés dans $\mathbb{P}[X]$) ou par un entier naturel. Nous remarquons que si nous évaluons la requête, nous obtenons pour le n-uplet (1,2) l'annotation $a + b$ et pour (3,4) l'annotation c lorsqu'on est dans $\mathbb{P}[X]$. Si nous évaluons a par 4, b par 3 et c par 6, nous retrouvons les annotations pour (1,2) et pour (3,4) dans le cas des entiers naturels.

Ainsi pour un semi-anneau K, annoter une base de données B par des éléments de K puis évaluer une requête q donne les mêmes annotations pour chaque n-uplet que si nous associons à chaque n-uplet de B une indéterminée puis évaluons q dans le cadre du semi-anneau $\mathbb{P}[X]$ et remplaçons les indéterminées par les valeurs correspondantes dans K [9].

Le tableau 4.29 propose un récapitulatif des semi-anneaux usuellement utilisés en pratique et les informations qu'ils apportent.

Semi-anneau	**Information**
Entiers naturels	Nombre d'occurrences d'un n-uplet Sémantique multi-ensembliste
Booléens	Présence ou absence d'une donnée Sémantique ensembliste
Booléens avec une indéterminée	Tables incomplètes
Semi-anneau de sécurité	Degré de confidentialité d'une donnée
Entiers tropicaux	Coût d'obtention d'une donnée
Polynômes à coefficients entiers	Cadre généralisateur des semi-anneaux

Table 4.29 – Semi-anneaux et caractéristiques

Pour vérifier que les structures algébriques sont bien des semi-anneaux et vérifient les propriétés d'intégrité, chacune des structures doit être une instance des enregistrements `CSR` et `Integ`. Je ne présente dans la suite que les preuves dans le cas où elles posent une difficulté particulière. Dans le cas contraire, seuls les éléments choisis pour représenter chacune des structures sont présentées.

Entiers naturels

La bibliothèque `N` est celle qui est choisie pour la représentation des entiers naturels sous forme binaire. Les éléments et les opérations sont déjà présents dans cette bibliothèque. La relation d'ordre usuelle `TN` est déjà définie dans Coccinelle et correspond à l'ordre croissant sur les entiers. Instancier l'enregistrement `CSR` par les entiers naturels revient à prouver le lemme suivant :

```
Lemma N_is_CSR : CSR N.zero N.one N.add N.mul TN.
```

La bibliothèque `Semi_Ring` confirme que les entiers naturels sont bien une instance de `CSR` et `Integ`.

Booléens

Je propose ici d'étudier pour les booléens les différentes opérations et les relations d'équivalences possibles donnant un semi-anneau. La première difficulté concerne me choix de la relation d'équivalence à utiliser. Il n'existe pas de relation d'équivalence usuelle entre les booléens $\top$ et $\bot$. Je propose donc de regarder quelle relation binaire interne entre ces deux valeurs sont des relations d'équivalences.

Une relation d'équivalence entre les éléments $\top$ et $\bot$ impose :

— $\top = \top$ (caractère réflexif de la relation d'équivalence)

— $\bot = \bot$ (idem)

On élimine la possibilité $\top = \bot$ qui n'a pas d'intérêt particulier.

Il y a donc deux relations d'équivalences possibles `Obool` (avec $\top < \bot$) et `Obool2` (avec $\top > \bot$).

Les opérations possibles et non triviales sont : le "ou" ($\vee$), le "et" ($\wedge$), l'égalité booléenne ($=_b$) et le "ou exclusif" ($\oplus$). Les opérations non triviales qui ne sont pas retenues ne peuvent vérifier la propriété d'associativité.

Parmi ces opérations, toutes les combinaisons pour le choix des opérations $+$ et $\times$ ne fonctionnent pas pour les raisons suivantes :

— $\forall a \in \texttt{Bool}, (\top \vee a) = a$ est faux donc $(\texttt{Bool}, \top, \vee)$ n'est pas un monoïde commutatif

— De même, $(\texttt{Bool}, \bot, \wedge)$, $(\texttt{Bool}, \bot, =_b)$ et $(\texttt{Bool}, \top, \oplus)$ ne sont pas des monoïdes commutatifs

— $=_b$ n'est pas distributif sur $\vee$:
$\bot =_b (\bot \vee \top) = \bot \neq \top = (\bot =_b \bot) \vee (\bot =_b \top)$

— De même, pour $\oplus$ sur $\wedge$

— le zéro et le un doivent être différents

La bibliothèque `SemiRing` propose donc les preuves que les structures suivantes sont bien des semi-anneaux :

1. $(\texttt{Bool}, \top, \bot, \wedge, \vee, \texttt{Obool})$
2. $(\texttt{Bool}, \top, \bot, \wedge, \vee, \texttt{Obool2})$
3. $(\texttt{Bool}, \bot, \top, \vee, \wedge, \texttt{Obool})$
4. $(\texttt{Bool}, \bot, \top, \vee, \wedge, \texttt{Obool2})$
5. $(\texttt{Bool}, \top, \bot, =_b, \vee, \texttt{Obool})$
6. $(\texttt{Bool}, \top, \bot, =_b, \vee, \texttt{Obool2})$
7. $(\texttt{Bool}, \bot, \top, \oplus, \wedge, \texttt{Obool})$
8. $(\texttt{Bool}, \bot, \top, \oplus, \wedge, \texttt{Obool2})$

Cependant, certaines de ces structures ne vérifient pas `Integ`. En effet :

— $(\bot =_b \bot) = \top$ mais $\top \neq \bot$

— $(\bot \oplus_b \top) = \top$ mais $\bot \neq \top$

Seuls les semi-anneaux $(\texttt{Bool}, \top, \bot, \wedge, \vee, \texttt{Obool})$, $(\texttt{Bool}, \top, \bot, \wedge, \vee, \texttt{Obool2})$, $(\texttt{Bool}, \bot, \top, \vee, \wedge, \texttt{Obool})$ et $(\texttt{Bool}, \bot, \top, \vee, \wedge, \texttt{Obool2})$ sont bien des instances de `Integ`.

Booléens avec indéterminée

Pour représenter l'incertitude de la présence d'une donnée dans une table, la structure des booléens est étendue avec une valeur "peut-être". Ces booléens se basent sur le type `bool3` qui en plus de `true` et `false` contient aussi la valeur d'indécision `unknown`.

Les opérations sont étendues de la façon suivante :

— pour le "et", un élément b et `unknown` donnent `false` si b est `false`, `unknown` sinon

— pour le "ou", un élément b et `unknown` donnent `true` si b est `true`, `unknown` sinon

La comparaison étudiée ici est l'extension de `Obool` avec l'élément `unknown` étant supérieur strictement aux deux autres éléments.

Je me suis restreinte à prouver que $(\texttt{Bool3}, \bot_3, \top_3, \vee_3, \wedge_3, \texttt{Obool3})$ vérifie bien les caractéristiques d'un semi-anneau commutatif et est une instance de `Integ`. Les autres cas sont similaires.

Semi-Anneau de sécurité

Le semi-anneau de sécurité a nécessité de créer un type inductif `Security` avec 6 degrés de sécurité :

```
Inductive Security :=
|OneS : Security (* Unclassifed *)
|Restr : Security (* Restricted *)
|Confid : Security (* Confidential *)
|Secret : Security (* Secret *)
|TopSec : Security (* Top secret *)
|ZeroS : Security. (* Never available *)
```

Ainsi qu'une fonction de comparaison `compareS` qui classe les degrés par ordre de restriction croissante :

`OneS` $<$ `Restr` $<$ `Confid` $<$ `Secret` $<$ `TopSec` $<$ `ZeroS`

Les opérations pour cette structure sont les fonctions `minSecu` et `maxSecu` définies à partir de la fonction `compareS`. Elles définissent le minimum et le maximum entre deux niveaux de confidentialité.

Les preuves pour ce semi-anneau sont directes grâce au nombre fini de valeurs. J'obtiens donc que le semi-anneau de sécurité est bien un semi-anneau :

```
Lemma Secu_is_CSR : CSR ZeroS OneS m∈ecu maxSecu OSecu.
```

et est bien une instance de `Integ`.

Entiers tropicaux

Dans le but de prouver que les entiers tropicaux sont bien un semi-anneau, je me suis intéressée à comprendre comment la structure et les opérations ont

été modifiées pour passer des entiers naturels $(\mathbb{N}, 0, 1, +, \times)$ aux entiers tropicaux $(\mathbb{N}^\infty, \infty, 0, min^\infty, +^\infty)$. Nous remarquons que :

- un élément (∞) est ajouté à l'ensemble et est l'élément neutre de l'opération 1
- l'élément neutre de l'opération 1 des entiers naturels devient l'élément neutre de l'opération 2
- l'addition est étendue à l'élément ∞ (tous les éléments additionnés à l'infini donne l'infini) et devient l'opération 2.
- nous pouvons définir l'opération 1 des entiers tropicaux si on prend la relation de comparaison usuelle sur les entiers naturels, l'étend à l'infini et prend le minimum entre deux éléments

Je propose ici de généraliser ces observations : à partir d'une structure S_1, si nous la modifions pour obtenir une structure S_2, S_2 est un semi-anneau commutatif et l'instanciation de S_1 par les entiers naturels nous donne que S_2 est l'ensemble des entiers tropicaux.

La structure minimale nécessaire pour S_1 est un monoïde commutatif (`M`, `zero`, `plus`, `OM`) qui vérifie la caractéristique suivante :

```
plus_compat_eq_lt : ∀ a1 a2 b,
  lt a1 a2 → (eq (plus b a1) (plus b a2) ⋁ lt (plus b a1) (plus b a2))
```

L'infériorité entre deux éléments est donc partiellement conservée lors d'une addition. On nomme cette structure `CMcompat`.

La construction de S_2 commence par l'ajout de l'élément supplémentaire (`Infinity`) à l'aide d'un type inductif :

```
Inductive MI :=
| Infinity : MI
| EltM : M → MI.
```

La fonction de comparaison est étendue comme suit :

```
Definition compMI a1 a2 :=
  match a1 with
  | Infinity ⇒
    match a2 with
    | Infinity ⇒ Eq
    | EltM b2 ⇒ Gt
    end
  | EltM b1 ⇒
    match a2 with
    | Infinity ⇒ Lt
    | EltM b2 ⇒ (Oeset.compare OM) b1 b2
    end
  end.
```

L'opération 2 (`mulMI`) est définie de manière similaire en étendant la fonction `plus`. L'opération 1 (`minMI`) compare deux éléments avec `compMI` et renvoie le plus petit des deux.

Pour prouver que cette structure est bien un semi-anneau commutatif, j'ai prouvé l'instanciation suivante :

Lemma MI_is_CSR : CSR Infinity (EltM zero) minMI mulMI OOMI.

Elle est aussi une instance de l'enregistrement `Integ`.

Il reste ainsi à prouver que les entiers sont bien une instance de la structure S_1 :

Lemma N_is_CMc : CMcompat N.zero N.add TN.

Les entiers tropicaux sont bien un semi-anneau communicatif vérifiant `Integ`.

4.3 . Les polynômes de provenance

4.3.1 . Intuition

La formalisation du semi-anneau des polynômes à coefficients entiers a nécessité un effort supplémentaire par rapport aux instances déjà proposées. Dans cette partie, nous considérons les polynômes avec plusieurs indéterminées. Pour formaliser la structure de polynômes, je me suis intéressée à la forme canonique des polynômes.

Définition 4.1: Polynômes à plusieurs indéterminées

Soit un ensemble A, soient les c_i des coefficients dans A, les $x_{(i,j)}$ des indéterminées et les $a_{(i,j)}$ des arités.
Nous définissons la forme canonique d'un polynôme à plusieurs indéterminées et à coefficients dans A comme suit :

$$\sum_{i=0,..,n} c_i(\Pi_{j=0,..,m_i} x_{(i,j)}^{a_{(i,j)}})$$

et les éléments du polynôme doivent vérifier les propriétés suivantes :

- $\forall i \in \{0,..,n\}, \forall j,j' \in \{0,..,m_i\}^2, j \neq j' \Rightarrow x_{(i,j)} \neq x_{(i,j')}$
- $\forall i,i' \in \{0,..,n\}^2, i \neq i' \Rightarrow \Pi_{j=0,..,m_i} x_{(i,j)}^{a_{(i,j)}} \neq \Pi_{j'=0,..,m_{i'}} x_{(i',j')}^{a_{(i',j')}}$

Sous la forme canonique, il est plus simple de voir qu'un polynôme peut être représenté par une fonction partielle du multi-ensemble des indéterminées vers les coefficients.

Par exemple, le polynôme $p = 3xy^2 + 94x^3z^4 + 4$ est représenté par la fonction :

$$\begin{aligned} \{\{Indeter\}\} &\xrightarrow{\mathsf{p}} \mathbb{N} \\ \{\{x:1, y:2\}\} &\longrightarrow 3 \\ \{\{x:3, z:4\}\} &\longrightarrow 94 \\ \{\{\}\} &\longrightarrow 4 \end{aligned}$$

De manière plus générale, pour un polynôme $P = \sum_{i=0,..,n} c_i(\Pi_{j=0,..,m_i} x_{(i,j)}^{a_{(i,j)}})$ dont les coefficients sont dans A, la fonction partielle correspondante est :

$$\{\{Indeter\}\} \xrightarrow{\mathbf{P}} A$$
$$\{\{x_{(i,1)} : a_{(i,1)}, .., x_{(i,m_i)} : a_{(i,m_i)}\}\} \longrightarrow c_i$$

On remarque que ces fonctions partielles sont définies seulement sur un nombre fini d'éléments de l'ensemble de départ.

Pour démontrer que les polynômes à coefficients entiers sont bien un semi-anneau à l'aide de cette représentation des polynômes, la première étape a été d'implémenter une bibliothèque pour les fonctions partielles sur des ensembles finis abstraits (**Sous-section 3.4.2**). Nous nous intéressons ici au module `Femap4` de la bibliothèque `FiniteMap` qui définit un ensemble d'opérations entre les fonctions partielles, notamment pour calculer l'addition et la multiplication de polynômes. Dans un deuxième temps, j'instancie les ensembles de départ et d'arrivée des fonctions partielles pour obtenir la structure des polynômes à plusieurs indéterminées. Nous pouvons alors prouver que les polynômes sont bien une instance de `CSR` et de `Integ`.

Instanciation vers les polynômes

Nous instancions ici l'enregistrement du module `Femap4` de la bibliothèque `FiniteMap` proposée dans la **Sous-section 3.4.2** pour obtenir des polynômes et prouver qu'ils constituent un semi-anneau.

L'opération `empty` correspond au polynôme nul. Le polynôme constant égal à `1` est formalisé à l'aide de la fonction `singleton` en prenant comme ensemble de départ l'ensemble vide et l'associant à `1`. La fonction `union` de `Femap4` permet de faire l'addition de deux polynômes. La fonction `transpo x y f` permet de formaliser la multiplication entre le polynôme formalisé par `f` et le monôme ayant pour coefficient `y` et indéterminées l'ensemble `x`. L'opération `mul_Pol` généralise la multiplication de deux polynômes.

L'ensemble des indéterminées est formalisé par le type `variable`, la relation d'équivalence qui lui est associée est `OX` et `BePV` est l'ensemble des multi-ensembles d'indéterminées. Ils sont introduits sous forme d'hypothèses.

```
Hypothesis variable : Set.
Hypothesis OX : Oeset.Rcd variable.
Hypothesis BePV : Febag.Rcd OX.
```

Les coefficients sont représentés initialement par un semi-anneau quelconque `R` qui est plus tard instancié par l'ensemble des entiers naturels.

```
Hypothesis R_is_CSR : CSR R zero one plus mul TR.
```

La dernière étape pour pouvoir instancier `Femap4` est de vérifier que l'ensemble des multi-ensembles des indéterminées est bien une instance de `CIO` : c'est le cas grâce aux propriétés de l'union de deux multi-ensembles.

```
Definition OEMon_is_CIO : CIO Febag.union OEMon.
Hypothesis FePol : (FePmap.Rcd TR zero plus mul OEMon_is_CIO).
```

Ainsi, nous obtenons l'instance `FePol` qui formalise en Coq les polynômes à multiples indéterminées et à coefficients dans R

Avec cette instance, nous pouvons définir les éléments remarquables et les opérations sur les polynômes :

```
Notation Pol_0 := (FePmap.empty FePol).
Notation Pol_1 := (FePmap.singleton FePol (emptysetBE) one).
Notation add_Pol := (FePmap.union FePol).
Notation mul_Pol := (FePmap.mul_Pol FePol).
```

Nous remarquons que d'autres formes sont possibles pour définir le polynôme `zero` comme par exemple `FePmap.singleton FePol (emptysetBE) zero`. De manière générale, chaque polynôme peut être représenté par des fonctions partielles différentes. Cependant, ces formes sont toutes équivalentes grâce à la propriété sur `equal`.

4.3.2 . Preuve du caractère de semi-anneau

La difficulté de la preuve que `FePol` est bien semi-anneau réside dans les propriétés impliquant la multiplication de polynômes. En effet, notre formalisation utilise la fonction `fold_left` et nous avons vu qu'effectuer une induction sur cette fonction est délicat dès que l'on cherche à sortir des informations de l'accumulateur.

Les preuve impactées par la multiplication sont :

— le caractère neutre du polynôme constant égal à un
— l'absorbance du polynôme nul
— la commutativité, l'associativité et la compatibilité de la multiplicationpar rapport à l'équivalence sur les polynômes
— la distributivité de la multiplication sur l'addition.

Si les preuves du caractère neutre et de l'absorbance n'ont comme difficulté que de réécrire les preuves en passant de `fold_left` à la fonction `fold_left_wacc`, les autres preuves ne sont pas aussi directes. Le point délicat de ces preuves est illustré par le cas de la commutativité. Il nous faut alors montrer l'équivalence entre :

```
coeff x (fold_left (fun a e ⇒
              union (transpo e (coeff e a2) a1) a)
            (Feset.elements (base a2)) empty)
```

et

```
coeff x (fold_left (fun a e ⇒
              union (transpo e (coeff e a1) a2) a)
            (Feset.elements (base a1)) empty))
```

Le réflexe serait de faire une induction sur les éléments de la base soit de `a1` soit de `a2` après avoir réécrit les fonctions `fold_left` en `fold_wacc`. Néanmoins, aucune des deux bases n'est prise en argument par la fonction `fold_left` des deux côtés de l'équivalence. Pour pouvoir faire une preuve par induction, il nous faut

avoir les éléments des bases de `a1` ou `a2` dans les deux parties de l'équivalence simultanément.

Pour pallier cette difficulté, je propose un lemme montrant l'équivalence des coefficients de la fonction `transpo` et d'une fonction `fold_left_wacc` :

```
Lemma coeff_transpo_alt : ∀ x e y a,
    eq TR (coeff x (transpo e y a))
    (fold_left_wacc plus
       (Feset.elements (base a)) zero
       (fun x0 : Mon ⇒
        if Oeset.eq_bool x (x0 unionBE e)
        then mul (coeff x0 a) y
        else zero)).
```

Cela revient à parcourir chaque multi-ensemble de variables `x0` contenu dans la base de a et vérifier si l'union de `x0` et e donne `x`, et le cas échéant, faire la multiplication entre le coefficient de `x0` et `y`. Nous notons ici qu'au maximum, un seul des multi-ensembles peut vérifier cette condition grâce à la condition `OEMon_is_CIO`.

Grâce à cette équivalence, nous pouvons effectuer une induction sur les éléments de la base de `a1` ou `a2`. Cette technique est utilisée dans les théorèmes de la commutativité, l'associativité, la compatibilité de la multiplication et la distributivité de la multiplication sur l'addition.

De plus, notons que d'autres lemmes de la bibliothèque `FoldFacts` ont été utilisés pour les différentes preuves impliquant une multiplication : certains lemmes très génériques comme `fold_left_wacc_eq_3` se retrouvent dans d'autres sections de cette thèse et d'autres sont spécifiquement proposés pour les preuves sur les polynômes comme `commut_fold_left_wacc_same_3`.

J'ai ainsi obtenu le théorème suivant :

```
Lemma R_pol_is_CSR : CSR Pol_0 Pol_1 add_Pol mul_Pol OEPol.
```

L'instanciation donnant les polynômes entiers à coefficients dans l'ensemble des entiers naturels est immédiate grâce au lemme `N_is_CSR`.

4.4 . Semi-modules

Nous rappelons ici la définition d'un K-semi-module.

Définition 4.1: K-Semi-module $(W, 0_W, \oplus_W, \otimes_W)$

Un K-semi-module $(W, 0_W, \oplus_W, \otimes_W)$ sur semi-anneau commutatif $(K, 0_K, 1_K, +_K, \times_K)$ est un ensemble W muni d'une loi de composition interne $\oplus_W$, d'un élément particulier 0_W et une opération binaire $\otimes_W$: $K \times W \to W$ vérifiant :

- $\forall k \in K,\ k \otimes_W 0_W = 0_W$
- $\forall w \in W,\ 0_K \otimes_W w = 0_W$
- $\forall w \in W,\ 1_K \otimes_W w = w$
- $\otimes_W$ est distributif à gauche sur $+_K$:

$\forall k \in K,\ \forall w_1,\ w_2 \in W^2,\ k \otimes_W (w_1 \oplus_W w_2) = k \otimes_W w_1 \oplus_W k \otimes_W w_2$

— $\otimes_W$ est distributif à droite sur $+_K$:
$\forall k \in K,\ \forall w_1,\ w_2 \in W^2, (k_1 +_K k_2) \otimes_W w = k_1 \otimes_W w \oplus_W k_2 \otimes_W w$

— $\forall k_1,\ k_2 \in K^2,\ \forall w \in W,\ (k_1 \times_K k_2) \otimes_W w = k_1 \otimes_W (k_2 \otimes_W w)$

Comme pour les semi-anneaux, les semi-modules sont représentés par un enregistrement :

```
Record SemiModule (K W : Type) (zeroK oneK : K) (plusK mulK : K → K → K)
         (zeroW : W) (plusW :W → W → W) (tensW : K → W → W)
         (OK : Oeset.Rcd K) (OW : Oeset.Rcd W) :=
mk_SM
  {
    isCSR : CSR zeroK oneK plusK mulK OK;
    isCM : CM zeroW plusW OW;
    plus_distr_w : ∀ k w1 w2,
     eq OW (tensW k (plusW w1 w2)) (plusW (tensW k w1) (tensW k w2));
    absorb_zero_W : ∀ k, eq OW (tensW k zeroW) zeroW;
    plus_distr_k : ∀ k1 k2 w,
     eq OW (tensW (plusK k1 k2) w) (plusW (tensW k1 w) (tensW k2 w));
    absorb_zero_K : ∀ w, eq OW (tensW zeroK w) zeroW;
    mul_assoc_K : ∀ k1 k2 w,
     eq OW (tensW (mulK k1 k2) w) (tensW k1 (tensW k2 w));
    one_neutral_K : ∀ w, eq OW (tensW oneK w) w
  }.
```

Je ne m'attarde pas ici sur les preuves mais nous avons comme résultat que pour toute instance (`M`, `zero`, `plus`, `OM:Oeset.Rcd M`) de `CM`, `M` est un `N`-semi-module et `Bool`-semi-module comme défini dans [10].

4.5 . Formalisation abstraite de la how-provenance

Dans cette section, je formalise de façon abstraite dans l'assistant de preuve Coq la sémantique de la how-provenance pour l'algèbre relationnelle positive étendue aux expressions d'attributs et aux fonctions d'agrégation. Les choix faits pour cette formalisation ont été orientés par le souhait d'obtenir une sémantique exécutable après instanciation et le fait que les objets manipulés sont munis de relations d'équivalence.

Une variation de cette sémantique restreinte à l'algèbre relationnelle positive a été implémentée mais ne sera pas présentée ici. Cette dernière est équivalente à la sémantique étendue dans le cas des requêtes de l'algèbre relationnelle positive.

La première étape de la formalisation de la sémantique de la K-algèbre étendue est de poser en hypothèse le semi-anneau utilisé pour les annotations. Comme nous sommes dans la partie abstraite de la formalisation, nous prenons un semi-anneau abstrait :

```
Hypothesis K : Type.
Hypothesis zeroK oneK : K.
```

```
Hypothesis plusK mulK : K → K → K.
Hypothesis OK : Oeset.Rcd K.
Hypothesis SR : SemiRing.CSR zeroK oneK plusK mulK OK.
```

4.5.1 . K-relations

A première vue, nous voudrions formaliser les K-relations en Coq par une fonction de type `tuple -> K`. Cependant, nous cherchons avec cette formalisation à pouvoir exécuter le code et calculer à chaque étape d'une requête les annotations. Si on prend l'opérateur de projection, en ayant une fonction des n-uplets vers les annotations, il faudra parcourir tous les n-uplets dont la projection donne notre n-uplet d'intérêt et ce pour chaque n-uplet d'intérêt. De plus, on rappelle qu'une K-relation est une fonction quasi-nulle, c'est-à-dire qu'elle n'a que très peu d'annotations non nulles car une table est un multi-ensemble fini.

Pour représenter les K-relations, j'ai choisi d'utiliser un enregistrement qui contient la fonction des n-uplets vers les annotations et un champ contenant le (sur-)ensemble fini des n-uplets dont les annotations sont non nulles. Le dernier champ de cet enregistrement est la sorte de la K-relation : cela impose que les n-uplets aux annotations non nulles ont bien la même sorte.

Les K-relations sont donc représentées par :

```
Record Krel : Type :=
 mk_kr
  {
    f : tuple → K;
    support : setT;
    sort_krel : Fset.set (A T);
  }.
```

où `T` correspond à la structure de données de type `Tuple.Rcd`.

On dira que pour la K-relation `kr` :

— son champ `f` est la fonction des annotations de `kr`

— son champ `support` est le support de `kr`

— son champ `sort_krel` est la sorte de `kr`

Avec la structure actuelle, il n'y a aucune assurance que les K-relations vérifient les propriétés abordées précédemment :

— pour tout n-uplet t n'étant pas dans l'ensemble `support` d'une K-relation, l'annotation de t est nulle pour cette K-relation

— pour tout n-uplet t étant dans le `support` d'une K-relation, la sorte de t est égale à l'ensemble `sort_krel` de cette K-relation

J'ai donc un deuxième enregistrement `Krel_inv` contenant ces propriétés.

```
Record Krel_inv (kr : Krel) : Prop :=
  mk_kri
    {
     finite_support : ∀ t, Fset.mem t (support kr) = false → eq (f kr t) zeroK;
     sort_labels_supp : ∀ t, t ∈ (support kr) → labels T t ≡ sort_krel kr;
    }.
```

Les deux propriétés constituent des invariants que les K-relations se doivent de vérifier pour être considérées comme bien formées. Il est nécessaire de définir ces restrictions qui semblent évidentes pour les preuves papier mais doivent être précisées en Coq.

On note que contrairement à d'autres structures qui contiennent les types et les propriétés dans le même enregistrement, j'ai volontairement séparé les deux en deux enregistrements pour faciliter les preuves et la manipulation des K-relations.

4.5.2 . Environnements

Les environnements de la bibliothèque Datacert sont modifiés pour prendre en compte les annotations de chaque n-uplet en plus des informations présentes dans les environnements de Datacert. Le type des environnements devient donc une liste de couples contenant les groupements à effectuer et une K-relation :

```
Definition env_slice_prov := group_by * Krel.
Definition env_prov := list env_slice_prov.
```

Les fonctions `env_t` et `env_g` (définies dans la **Sous-section 3.3.1**) sont elles aussi modifiées pour s'adapter à la nouvelle structure des environnements dans le cadre de la provenance.

Il est souhaitable que les différentes façons de définir un environnement n'influencent pas les résultats des fonctions manipulant des environnements. La propriété à prendre en compte pour la définition de nos fonctions est que si deux environnements sont équivalents, les résultats des fonctions doivent être équivalents.

J'ai donc défini une relation d'équivalence entre deux environnements :

```
Definition equiv_env_slicep (e1 e2 : env_slice_prov) :=
let (g1, kr1) := e1 in
let (g2, kr2) := e2 in
  sort_krel kr1 ≡sort_krel kr2 ⋀
  (filter_zero kr1 =SE= filter_zero kr2) ⋀
  (∀ t1 t2, eq_bool t1 t2 = true → eq (f kr1 t1) (f kr2 t2)) ⋀
  g1 = g2.

Definition equiv_envp (e1 e2 : env_prov) := Forall2 equiv_env_slicep e1 e2.
```

Deux environnements sont équivalents si :

— pour toutes les K-relations prises deux à deux, leurs ensembles de n-uplets à annotations non nulles sont équivalents, leurs sortes sont équivalentes et les fonctions d'annotations renvoient les mêmes annotations à n-uplet équivalent près

— les groupements sont les mêmes

4.5.3 . Évaluation des requêtes

La première étape dans l'évaluation des requêtes est de poser en hypothèse les informations relatives aux relations sources.

```
Hypothesis instance_prov : relname → Krel.
Hypothesis instance_prov_equiv : ∀ r x1 x2,
 eq OTupleT x1 x2 → eq OK (f (instance_prov r) x1) (f (instance_prov r) x2).
Hypothesis instance_prov_inv : ∀ r, Krel_inv (instance_prov r).
```

`instance_prov` associe à chaque nom de relation source une K-relation. Pour que les K-relations sources vérifient les invariants de `Krel_inv` et que la fonction des annotations de celles-ci soient compatible avec l'équivalence des n-uplets, j'ai posé les hypothèses `instance_prov_equiv` et `instance_prov_inv`.

Chacun des autres opérateurs est défini par une fonction prenant en arguments les résultats des sous-requêtes, un environnement et les arguments de l'opérateur et renvoie la K-relation correspondant à cet opérateur.

Par exemple, pour la sélection, nous avons une fonction :

```
Definition Krel_sigma (kr : Krel) (env : env_prov) (form : formula) :=
  mk_kr (f_sigma kr env form) (support kr) (sort_krel kr).
```

où `kr` correspond au résultat de la sous-requête, `env` l'environnement et `form` la formule sur laquelle se fait la sélection. Nous remarquons ici qu'on instancie l'enregistrement `Krel` grâce à `mk_kr`.

Le **Table 4.30** détaille l'implémentation des différents champs de ces fonctions pour les opérateurs de l'algèbre relationnelle avec expressions d'attributs. Le cas des fonctions d'agrégation est détaillé dans une seconde partie.

Op	Fonction (`fun t => ..`)	Support	Sorte
ϵ	`if Oeset.eq_bool t empty_tuple` `then oneK else zeroK`	`Feset.singleton empty_tuple`	`emptysetS`
r	`f (instance_prov r) t`	`support (instance_prov r)`	`basesort r`

pi_s	`fold_left`
	`(fun x y => plusK (f kr y) x)`
	`(filter`
	`(fun t' => Oeset.eq_bool OTupleT t`
	`(projection_prov (env_pt env t') s))`
	`(Feset.elements (support kr))))`
	`zeroK)`
		`Feset.map`
		`(fun t => projection_prov (env_pt env t) s)`
		`(support kr)`
			`Fset.mk_set s`

σ_{form}	`if is_true (eval_formula_prov (env_pt env t) form)` `then f kr t else zeroK`	`support kr`	`sort_krel kr`

$\bowtie$	`if labels T t =S?= (sort_krel kr1 unionS sort_krel kr2)`
	`then mulK`
	`(f kr1 (mk_tuple (sort_krel kr1) (dot t))`
	`(f kr2 (mk_tuple (sort_krel kr2) (dot t))`
	`else zeroK`
		`Feset.mk_set`
		`(natural_join_list`
		`(Feset.elements (support kr1))`
		`(Feset.elements (support kr2)))`
			`sort_krel kr1`
			`unionS`
			`sort_krel kr2`

$\cup$	`if sort_krel kr1 =S?= sort_krel kr2`
	`then plusK (f kr1 t) (f kr2 t)`
	`else zero`
		`if sort_krel kr1 =S?= sort_krel kr2`
		`then Feset.union (support kr1) (support kr2)`
		`else Feset.empty FesetT`
			`sort_krel kr1`

Table 4.30 – Formalisation des opérateurs sauf fonctions d'agrégation

Les `kr(i)` désignent les résultats des évaluations des sous-requêtes, `env` l'environnement sur lequel la requête est évaluée, `s` correspond à une fonction de renommage et `form` à une formule. La fonction `eval_formula_prov` évalue la formule logique sur un environnement.

Remarque
Les fonctions des annotations utilisent bien le support pour calculer certaines annotations. La fonction `f` prend la plupart du temps comme argument le n-uplet t mais pour le cas de la jointure naturelle, il est nécessaire de restreindre le n-uplet sur les attributs des sous-requêtes donc sur `sort_krel kri` (où `i` correspond à 1 ou 2). De plus, comme nous sommes dans le cas d'une formalisation en Coq, certaines fonctions des annotations et certains supports nécessitent de vérifier la sorte des sous-requêtes pour que les preuves soient réalisables.

Nous nous attardons sur la fonction `projection_prov`.

```
Definition projection_prov (env : env_prov) (las : list select) :=
  mk_tuple
    (Fset.mk_set (map (fun x ⇒ match x with
                               | Select_As _a ⇒ a
                               end) las)
    (fun a ⇒
     match Oset.find a las with
     | Some e ⇒ interp_aggterm_prov env e
     | None ⇒ dot (default_tuple (∅)) a
     end).
```

La fonction `projection_prov` prend un environnement et une fonction de renommage. L'environnement contient en pratique soit la K-relation correspondant au n-uplet sur lequel il faut appliquer la fonction de renommage `las` soit celle d'un groupe de n-uplet sur lequel nous appliquons une fonction d'agrégation. La fonction `mk_tuple` produit un n-uplet t dont la sorte est prise en premier argument et dont ses valeurs sont données par une fonction prise en second argument qui associe à chaque attribut, la valeur de t pour cet attribut. `projection_prov` renvoie un n-uplet dont les valeurs pour chaque colonne sont données par `interp_aggterm_prov` qui calcule soit la projection soit la fonction d'agrégation sur les éléments de l'environnement.

Dans cette partie, comme nous nous intéressons à l'algèbre relationnelle positive étendue, il n'est pas nécessaire de porter une attention particulière aux fonctions de renommage et à la sorte des n-uplets projetés et renommés (de même pour les fonctions d'agrégation). La sorte d'un tel n-uplet correspond à l'ensemble des attributs en sortie de la fonction de renommage.

Cependant, dans le contexte de l'algèbre relationnelle positive sans extension, la fonction `projection_prov` renvoie des n-uplets dont la sorte garde seulement les attributs en sortie de la fonction de renommage qui avant d'être renommé appartenait à la sorte du n-uplet initial. Pour la sémantique de la how-provenance restreinte à l'algèbre relationnelle positive, la fonction `projection_prov` est modifiée pour prendre cela en compte.

Les sémantiques obtenues dans les deux cas restent équivalentes si nous nous restreignons à l'algèbre relationnelle positive et pour chaque opérateur η, nous choisissons des fonctions de renommage dont l'ensemble d'entrée est strictement

contenu dans la sorte de la sous-requête. Cette variante permet dans les chapitres suivants de confronter la sémantique de la how-provenance à celles de la where-provenance et la wow-provenance qui sont définies sur l'algèbre relationnelle positive.

Fonctions d'agrégation

La formalisation en Coq des K-relations associées à une fonction d'agrégation nécessite l'ajout en hypothèse de la fonction abstraite δ.

```
Hypothesis delta_f : list K → K.
```

Ici la fonction englobe aussi la somme entre les annotations donc le type de l'argument de δ est une liste d'annotations.

La découpe (ou non) en groupes des n-uplets ayant une annotation non nulle d'une K-relation se fait à l'aide de la fonction `mk_groups_prov` selon les valeurs prises par ces n-uplets sur les expressions d'attributs du groupement.

```
Definition make_groups_prov env kr gby :=
match gby with
| Group_By g ⇒
    map snd
        (partition (fun t ⇒ map (fun agg ⇒
            interp_aggterm_prov
                (env_gp env (Group_By g)
                            (mk_kr (fun t0 ⇒ if Oeset.eq_bool t0 t then oneK
                                                                   else zeroK)
                (Feset.singleton t) (sort_krel kr))) agg) g) kr)
end.
```

La fonction `partition` prend en argument une fonction qui interprète les ensembles d'expressions d'attributs de g sur un n-uplet et la K-relation devant être divisée en groupes. Elle renvoie une liste de couples avec les valeurs prises par les n-uplets du groupe et la K-relation restreinte à ce groupe.

La fonction d'interprétation `interp_aggterm_prov` ici est la même que celle contenue dans la fonction `projection_prov`. L'environnement contient le n-uplet à interpréter sur les expressions d'attributs données en deuxième argument de la fonction. Ici le n-uplet à interpréter doit être converti en K-relation pour être au format attendu pour un environnement.

Une fois la K-relation divisée en groupe de K-relations, il faut appliquer les fonctions d'agrégations et les expressions d'attributs sur les ensembles de n-uplets pour obtenir les n-uplets résultants de la requête. Cela se fait avec la fonction `projection_prov`, ce qui nous donne la fonction `calcul_proj_grb`.

```
Definition calcul_proj_grb env lf s kr :=
  map (fun l ⇒ (projection_prov (env_gp env (Group_By lf) l) s), l))
                    (make_groups_prov env kr (Group_By lf)).
```

Le calcul des annotations de chaque n-uplet se fait à l'aide de la fonction `f_gamma` qui a pour rôle d'appliquer la fonction δ sur les annotations et renvoyer la fonction des annotations.

```
Fixpoint f_gamma l :=
  match l with
  |nil ⇒ fun t ⇒ zeroK
  |(t0,kr) :: tl ⇒ let f1 := f_gamma tl in
   fun t ⇒ if Oeset.eq_bool t0 t
            then plusK (delta_f (map (f kr) (Feset.elements (support kr)))) (f1 t)
            else f1 t
  end.
```

La fonction donnant la K-relation associée aux fonctions d'agrégation est donc la suivante, qui utilise les fonctions précédentes.

```
Definition Krel_gamma l env kr lf :=
    let lproj := calcul_proj_grb env lf l kr in
    let supp := Feset.mk_set (map fst lproj) in
    let attr := Fset.mk_set (map (fun x ⇒ match x with
                                          | Select_As _a1 ⇒ a1
                                          end) l) in
    mk_kr (f_gamma lproj) supp attr
```

Nous remarquons que la sorte de la K-relation est obtenue en récupérant les attributs en sortie de la fonction de renommage.

L'évaluation complète d'une requête s'effectue à l'aide de la fonction `eval_prov_k` qui applique la récursion sur les sous-requêtes.

4.5.4 . Sorte

La fonction sur la sorte d'une requête n'est pas affectée par nos restrictions sur l'expressivité des requêtes et l'ajout des annotations. Elle reste donc la même : `sort`.

4.5.5 . Bonne formation

Nous redéfinissons la fonction de bonne formation donnée par Datacert pour deux raisons. La première est que celle-ci ne prend pas en compte notre restriction de l'expressivité des fonctions d'agrégation qui ne doivent pas apparaître dans des sous-requêtes. La seconde s'explique par le fait que les fonctions d'agrégation utilisables pour notre formalisation doivent avoir un domaine qui forme un semi-module et donc doivent vérifier une condition supplémentaire.

La fonction `well_formed` prend en argument un environnement, une variable booléenne `b` et une requête et renvoie un booléen.

La variable booléenne assure que l'opérateur Γ ne peut être qu'en tête de la requête et n'apparaît dans aucune sous-requête. `b` est initialisée à `true` et la variable est passée à `false` lors de la première itération de `well_formed`. Si on initialise le booléen à `false`, on se place dans l'algèbre relationnelle étendue aux expressions d'attributs.

Pour ce qui est de la bonne formation des fonctions d'agrégation, le type de ces dernières étant abstrait, je pose en hypothèse la fonction `wf_aggregate` qui vérifie qu'une fonction d'agrégation est bien formée. La fonction `well_formed_aggregate`

applique cette fonction à chaque élément de l'ensemble des attributs en entrée de la fonction de renommage.

En reprenant les définitions de la bonne formation des requêtes évoquées dans les **Sous-sections 3.1.2, 3.2.1 et 3.2.2**, nous obtenons la fonction suivante :

```
Fixpoint well_formed env b q :=
  match q with
  | Q_Empty_Tuple ⇒ true
  | Q_Table r ⇒ true
  | Q_Union q1 q2 ⇒
    if sort (to_sql_query q1) =? sort (to_sql_query q2)
    then well_formed env false q1 && well_formed env false q2
    else false
  | Q_NaturalJoin q1 q2 ⇒
    well_formed env false q1 && well_formed env false q2
  | Q_Pi s q ⇒
        well_formed env false q
        && well_formed_s (env_t env (default_tuple (sort q))) s
        && is_not_aggregate s
  | Q_Sigma f q ⇒
    well_formed env false q && well_formed_form env f
  | Q_Gamma s g q ⇒
    b && well_formed env false q && well_formed_aggregate s
    && well_formed_s (env_g env (Group_By g) (default_tuple (sort q) :: nil)) s
    && forallb (well_formed_ag (env_t env (default_tuple (sort q)))) g
  end
```

La bonne formation des relations sources est assurée par le type `relname` qui correspond à l'ensemble des noms de celles-ci. Celle de l'union de deux sous-requêtes dépend de l'équivalence entre les sortes des ces sous-requêtes.

Dans le cadre de l'opérateur de la projection et du renommage, il est nécessaire de limiter l'argument correspondant à la fonction de renommage car le type choisi inclut des fonctions d'agrégation. C'est le rôle de la fonction `is_not_aggregate`. Il en est de même pour l'argument de groupage.

La fonction `well_formed_s` est hérité de Datacert et vérifie que que les attributs en entrée sont bien dans la sorte de la sous-requête et les attributs apparaissant en sortie de la fonction de renommage sont deux à deux distincts.

On notera que l'environnement ici est celui de Datacert et non de notre formalisation actuelle. Cela nous facilite la comparaison entre la sémantique de Datacert et celle de la how-provenance. Une autre raison est que les annotations ne sont pas impliquées dans la bonne formation d'une requête et nous aurions à redéfinir des fonctions déjà existantes pour qu'elles soient compatibles avec les environnements avec annotations, ce qui est contre-productif.

Certaines des preuves qui suivent se contentent d'une fonction allégée de bonne formation `well_formed_query_agg` qui ne prend en compte que certains éléments de la fonction `well_formed`.

4.5.6 . Théorèmes généraux et invariants

Pour valider la conformité de la formalisation des K-relations et de la sémantique de la how-provenance avec les attentes formulées dans la section précédente, je propose dans cette section de vérifier que chaque K-relation issue de l'évaluation d'une requête vérifie les propriétés suivantes :

1. **(Non appartenance - annotation nulle)** Si un n-uplet n'est pas dans le support d'une K-relation, alors l'annotation associée à ce n-uplet par la fonction des annotations de cette K-relation est nulle.
2. **(Equivalence sorte n-uplet-K-relation)** Si un n-uplet est dans le support d'une K-relation alors la sorte de ce n-uplet est la sorte de la K-relation.
3. **(Equivalence sorte requête-K-relation)** La sorte d'une requête est égale à la sorte de la K-relation résultant de l'application de cette requête.
4. **(Compatibilité avec les n-uplets)** Les évaluations de la fonction des annotations pour deux n-uplets équivalents renvoient des résultats équivalents. Il en va de même pour les fonctions `supp` et `sort_krel`.
5. **(Compatibilité avec les environnements)** Les évaluations de la fonction des annotations pour deux environnements équivalents renvoient des résultats équivalents. Il en va de même pour les fonctions `supp`, `sort_krel` et `eval_formula_prov`.

Pour cela, j'ai prouvé cinq théorèmes dans l'assistant de preuve Coq.

Les deux premières propriétés nécessitent de prouver que chaque K-relation résultant d'une requête soit une instance de `Krel_inv`.

```
Theorem Krel_inv_eval_k : ∀ env q,
    Krel_inv (eval_prov_k env q).
```

Nous remarquons que le support est bien une sur-approximation de l'ensemble des n-uplets d'annotation non nulle avec le champ `finite_support`.

La troisième propriété est prouvée par le théorème suivant :

```
Theorem sort_krel_sort_N : ∀ env q,
    sort q = sort_krel (eval_prov_k env q).
```

Comme la fonction `sort` ne dépend pas de l'environnement, nous avons la compatibilité de `sort_krel` avec l'équivalence des environnements.

Pour ce qui est des formules, la vérification de leur comptabilité avec les environnements est obtenue par :

```
Theorem eval_formula_eq_env : ∀ form env1 env2,
    equiv_envp env1 env2 →
    eval_formula_prov env1 form = eval_formula_prov env2 form.
```

Ces trois premiers théorèmes ne posent pas de difficulté particulière et la preuve est une preuve inductive sur la structure de la requête ou de la formule.

Les deux dernières propriétés pour le champ `f` et `support` sont regroupées dans un même lemme et sont prouvées simultanément à cause des inter-dépendances entre ces deux champs. Pour des raisons similaires, la compatibilité avec l'équivalence entre n-uplets et celle entre les environnements sont prouvées conjointement.

```
Lemma eval_prov_eq_env_tuple_aux : ∀ n,
    (∀ b q env1 env2,
        tree_size (tree_of_query q) ⩽ n →
        well_formed_query_agg b q = true →
        equiv_envp env1 env2 →
        support (eval_prov_k env1 q) =SE= support (eval_prov_k env2 q)) ⋀
      (∀ b q env1 env2,
        tree_size (tree_of_query q) ⩽ n →
        well_formed_query_agg b q = true →
        equiv_envp env1 env2 →
        ∀ t1 t2,
          eq OTupleT t1 t2 →
          eq OK (f (eval_prov_k env1 q) t1) (f (eval_prov_k env2 q) t2)).
```

La preuve est ici faite par induction sur la taille de la requête et non la requête elle-même. Pour cela, on pose en hypothèse la supposition que la taille de la requête est inférieur à `n` et effectue une induction sur cet entier.

```
tree_size (tree_of_query q) ⩽ n
```

La taille de la requête correspond à la taille de l'arbre syntaxique de la requête. Cet arbre est obtenu grâce à la fonction `tree_of_query` et la fonction `tree_size` renvoie la taille de l'arbre.

Ce dernier lemme nécessite l'utilisation des théorèmes `sort_krel_sort_N` et `eval_formula_eq_env` dans sa preuve.

Il découle de ce lemme, les deux théorèmes suivants :

```
Theorem f_eval_prov_eq_env_tuple : ∀ b q env1 env2 t1 t2,
    well_formed_query_agg b q = true →
    equiv_envp env1 env2 →
    eq OTupleT t1 t2 →
    eq OK (f (eval_prov_k env1 q) t1) (f (eval_prov_k env2 q) t2).
```

et :

```
Theorem support_eval_prov_eq_env_tuple : ∀ b q env1 env2,
    well_formed_query_agg b q = true →
    equiv_envp env1 env2 →
    support (eval_prov_k env1 q) =SE= support (eval_prov_k env2 q).
```

On notera aussi que les sous-fonctions utilisées pour définir `eval_prov_k` sont elles aussi compatibles à environnements et n-uplets équivalents près, notamment la fonction `projection_prov` avec le lemme `projection_prov_eq` que nous utiliserons dans la suite.

4.6 . Equivalence avec la sémantique de Datacert

Pour valider le modèle abstrait que nous venons de détailler, je propose dans cette section et la suivante des instanciations de la sémantique de la how-provenance. L'instanciation de cette section se concentre sur le choix du semi-anneau commutatif : nous prenons pour les annotations l'ensemble des entiers naturels.

Dans la **Section 4.1**, nous avons vu qu'une relation est représentable par une fonction des n-uplets vers son nombre d'occurrences, ce qui correspond à annoter chaque n-uplet source par son nombre d'occurrences et à suivre la propagation de ces annotations. Après application d'une telle requête, nous devrions avoir comme annotations le nombre d'occurrences des n-uplets résultants.

Pour vérifier cela, je propose dans cette section de montrer en Coq le théorème d'adéquation :

Théorème 4.1: Adéquation

Si pour tout n-uplet d'une table source, l'annotation de ce n-uplet est égale à son nombre d'occurrences, alors pour toute requête, l'annotation d'un n-uplet résultant est égale au nombre d'occurrences de ce n-uplet.

$$\begin{aligned}&(\forall r \in \mathcal{N}rel_s,\\ &\qquad \forall t \in \mathcal{T}, [\![r]\!]^{\mathbb{N}}_{hw}(t) = \boldsymbol{nb}([\![r]\!]_{rel}, t))) \Rightarrow\\ &\forall q,\\ &\qquad \forall t \in \mathcal{T}, [\![q]\!]^{\mathbb{N}}_{hw}(t) = \boldsymbol{nb}([\![q]\!]_{rel}, t))\end{aligned}$$

Ce théorème nécessite de comparer la sémantique pour l'algèbre relationnelle positive étendue de la how-provenance et la sémantique multi-ensembliste proposée dans la bibliothèque Datacert. Nous cherchons donc à avoir un lemme de la forme :

```
f (eval_prov_k (K := N) env q) t = Febag.nb_occ t (eval_query_rel env' q).
```

Avant de se lancer dans la preuve de ce théorème, j'ai :

1. défini des tables sources non annotées (multi-ensembles).
2. annoté chaque n-uplet de ces tables avec son nombre d'occurrences
3. instancié la fonction `delta_f` avec la fonction δ_N
4. instancié les fonctions et lemmes précédents avec le semi-anneaux des entiers.

Pour les tables sources, cela revient à poser en hypothèse :

```
Hypothesis instance_rel : relname → bagT.
```

qui associe au nom de la relation source son multi-ensemble de n-uplets.

Comme précédemment, j'utilise la fonction `instance_prov` pour manipuler les tables sources avec leurs annotations. On remarque ici que les annotations sont dans l'ensemble des entiers naturels `N`.

```
Hypothesis instance_prov : relname → Krel N.
Hypothesis instance_prov_nb_occ : ∀ r e,
    f (instance_prov r) e = Febag.nb_occ BTupleT e (instance_rel r).
```

L'hypothèse supplémentaire `instance_prov_nb_occ` assure que les annotations des n-uplets des tables sources correspondent bien aux nombres d'occurrences de ces n-uplets.

Pour la fonction δ_N, nous avons une fonction récursive qui parcourt les annotations est renvoie `0` si toutes les annotations sont nulles et `1` sinon.

```
Fixpoint delta_f (l : list N) : N :=
match l with
|nil ⇒ 0
|hd :: tl ⇒ if Oeset.eq_bool TN hd 0 then delta_f tl
            else 1
end.
```

Je ne détaille pas toutes les instanciations des fonctions et des lemmes par les entiers.

Hypothèses supplémentaires

Dans le cas des annotations dans $\mathbb{N}$, la formalisation dans un assistant de preuve des fonctions d'agrégation nécessite d'expliciter certaines hypothèses faites sur les fonctions les évaluant sur des valeurs. En pratique, ces dernières correspondent aux valeurs prises par les n-uplets d'un groupe sur l'attribut de la fonction d'agrégation.

Les fonctions dont nous parlons sont la fonction abstraite `interp_aggregate` pour Datacert et la fonction abstraite `interp_aggregate_prov` pour la how-provenance. Les fonctions d'évaluation des fonctions d'agrégation prennent en argument le nom de la fonction d'agrégation $agg(a)$ à appliquer et une liste. Pour `interp_aggregate`, la liste est constituée des valeurs prises par les n-uplets sur l'attribut a sur lequel on applique agg et la liste répète une valeur autant de fois que son n-uplet associé apparaît dans la table. Pour `interp_aggregate_prov`, la liste contient des couples de valeurs prises par les n-uplets et le nombre d'occurrences associé à ces n-uplets dans la table. La liste est donc aussi longue qu'il y a de n-uplets différents. Il est cependant possible que deux n-uplets différents soient associés à la même valeur et donc que la valeur apparaisse plusieurs fois. Ces deux fonctions renvoient l'application de a sur les valeurs.

Il faut ainsi imposer que ces deux fonctions d'interprétation coïncident lorsque `interp_aggregate_prov` prend en argument une liste `l` dont toutes les valeurs d'entiers sont égales à 1 et que `interp_aggregate` prend l sans les entiers associés :

```
Hypothesis interp_aggregate_prov_prop1 : ∀ a l,
    wf_aggregate a = true →
    List.forallb  (fun x ⇒ Oeset.eq_bool (snd x) 1) l = true →
    interp_aggregate_prov a l = interp_aggregate a (map fst l).
```

De plus, nous constatons par exemple que la fonction `interp_aggregate_prov` doit donner le même résultat pour les listes `[(v1,n1), (v2, n2), (v1, n3), (v2,0)]` et `[(v1,n1 + n3), (v2, n2)]` car les valeurs `v1` et `v2` apparaissent

autant de fois dans les deux listes. Je généralise cela avec une deuxième hypothèse : la fonction `interp_aggregate_prov` donne le même résultat pour deux listes différentes `l1` et `l2` si pour chaque valeur v, on obtient le même nombre lorsqu'on somme tous les entiers associés à v de la liste `l1` ou de la liste `l2`.

```
Definition sum l := fold_left N.add l 0.

Definition filter_sum y l := sum (map snd (filter (fun x ⇒ Oset.eq_bool (fst x) y) l)).

Hypothesis interp_aggregate_prov_prop2 : ∀ a l1 l2,
            wf_aggregate a = true →
            (∀ t, filter_sum t l1 = filter_sum t l2) →
            interp_aggregate_prov a l1 = interp_aggregate_prov a l2.
```

Enfin, il est important de souligner que pour prouver nos équivalences entre les deux sémantiques, il est nécessaire de pouvoir passer d'un environnement de Datacert à un environnement avec les annotations. Dans notre cas, les annotations correspondent aux nombres d'occurrences d'un n-uplet dans une tranche de l'environnement. La fonction `create_env` remplit cet objectif.

Il nous reste maintenant à prouver le théorème d'adéquation :

```
Theorem K_relations_extend_relational_algebra :
  ∀ env (q : query) b,
    well_formed env b q = true →
    ∀ (t : tuple),
      f (eval_prov_N (create_env env) q) t =
      Febag.nb_occ t (eval_query_rel env q).
```

Nous avons besoin que la requête soit bien formée. La partie de l'égalité issue de Datacert est de couleur cyan et celle issue de la formalisation de la how-provenance est colorée en orange.

La preuve de ce théorème est faite par induction sur la taille de la requête. Elle se fait conjointement avec la preuve d'égalité entre les fonctions d'évaluation des formules de Datacert et celles de notre formalisation. Je détaille les preuves pour le cas de l'union, de la projection et de l'application de fonctions d'agrégation.

Union de deux requêtes

Prouver le cas de l'union entre deux requêtes revient à prouver l'égalité suivante :

```
 f (Krel_union (eval_prov_N (create_env env) q1)
               (eval_prov_N (create_env env) q2)) e =
Febag.nb_occ e
    (if sort q1 =?= sort q2
     then Febag.union (eval_query_rel env q1) (eval_query_rel env q2)
     else emptyset)
```

La définition de `Krel_union` de la partie de l'égalité colorée en orange se déroule en :

```
if sort_krel (eval_prov_N (create_env env) q1) =?=
  sort_krel (eval_prov_N (create_env env) q2)
```

```
then
    f (eval_prov_N (create_env env) q1) e +
    f (eval_prov_N (create_env env) q2) e
else 0
```

Nous remarquons déjà que nous avons en violet les fonctions donnant soit la sorte des requêtes q1 et q2, soit la sorte des K-relations associées aux requêtes q1 et q2. Avec le théorème **(Equivalence sorte requête-K-relation)** de la section précédente, les deux types de sortes sont égaux. Il nous reste donc à prouver :

```
if sort q1 =? sort q2
then
    f (eval_prov_N (create_env env) q1) e +
    f (eval_prov_N (create_env env) q2) e
else 0 =
Febag.nb_occ e
    (if sort q1 =? sort q2
     then Febag.union (eval_query_rel env q1) (eval_query_rel env q2)
     else emptyset)
```

Il suffit ici de faire deux cas : le cas où les sortes de q1 et q2 sont équivalentes et le cas où elles ne le sont pas. Pour le deuxième cas, la preuve est triviale par définition du multi-ensemble vide. Pour le premier cas, nous réécrivons la spécification du nombre d'occurrences de l'union entre deux multi-ensembles (en cyan) :

```
f (eval_prov_N (create_env env) q1) e + f (eval_prov_N (create_env env) q2) e =
Febag.nb_occ e (eval_query_rel env q1) + Febag.nb_occ e (eval_query_rel env q2)
```

On peut alors appliquer l'hypothèse d'induction sur q1 (en violet) et q2 (en orange) car leur taille est plus faible que la requête principale. Même si la preuve dans le cas de l'union de deux requêtes ne présente pas de difficulté particulière, l'emploi de la fonction `fold_left` dans les preuves des deux cas présentés ci-après a demandé un effort plus important.

Projection-renommage sur expressions d'attributs

Pour le cas de la projection-renommage sur des expressions d'attribut, le but de la preuve est le suivant (après quelques simplifications) :

```
fold_left
  (fun x y ⇒ f (eval_prov_N (create_env env) q) y + x)
  (filter
     (fun t ⇒
      Oeset.eq_bool e
        (projection_prov
           (env_tp (create_env env) t) l)))
     (Feset.elements (support (eval_prov_N (create_env env) q))))
    0 =
 Oeset.nb_occ e
    (map
       (fun t ⇒
```

```
      projection (env_t env t) l)
    (Febag.elements (eval_query_rel env (to_sql_query q))))
```

Nous cherchons ici à déterminer soit le nombre d'occurrences soit l'annotation du n-uplet `e`. La première partie de l'égalité correspond à celle de la sémantique de la how-provenance où chaque n-uplet du `support` de la sous-requête `q` est projeté-renommé selon `l` puis comparé à `e`. Les annotations des n-uplets projetés-renommés étant égaux à `e` sont sommées entre elles pour donner l'annotation de `e`. La deuxième partie de l'égalité correspond à la partie venant de Datacert. Les éléments du multi-ensemble associé à `q` sont projetés-renommés selon `l` à l'aide de la fonction `map`. On compte ensuite le nombre d'occurrences de `e` dans la liste résultant des éléments.

Le réflexe pour la première étape de cette preuve est d'utiliser la fonction alternative `fold_left_wacc` ainsi que les preuves proposées dans la bibliothèque `FoldFatcs` (**Sous-section 3.4.1**).

On passe donc d'une preuve sur `fold_left` (en orange) à une preuve sur `fold_left_wacc` avec la fonction `fold_left_wacc_equiv`.

```
fold_left_wacc N.add
    (filter
       (fun t ⇒
        Oeset.eq_bool OTupleT e
          (projection_prov (env_tp (create_env env) t) l)))
       (Feset.elements (support (eval_prov_N (create_env env) q))))
    0 (f (eval_prov_N (create_env env) q)) =
 Oeset.nb_occ e
    (map
       (fun t ⇒
        projection (env_t env t) l)
       (Febag.elements (eval_query_rel env (to_sql_query q))))
```

Afin de prouver le résultat, il est nécessaire de passer d'une preuve contenant des éléments spécifiques à la sémantique de la how-provenance (comme le `support`, la fonction `projection_prov` ou la fonction d'annotation `f`) à une preuve ne contenant que les éléments de la sémantique de Datacert. Pour ce faire, une méthode consiste à commencer par réécrire la fonction donnant les annotations de `q` par le nombre d'occurrences des éléments de `q` grâce à l'hypothèse d'induction. Pour appliquer cette hypothèse, j'utilise le lemme `fold_left_wacc_equiv_eq_3_2` qui permet de remplacer la fonction appliquée par `fold_left_wacc` par une autre fonction qui lui est équivalente (en violet) :

```
fold_left_wacc N.add
    (filter
       (fun t : tuple ⇒
        Oeset.eq_bool OTupleT e
          (projection_prov 0 1 TN interp_aggregate_prov
             (env_tp (create_env env) t) (Select_List (_Select_List l))))
       (Feset.elements FesetT (support (eval_prov_N (create_env env) q))))
    0
```

```
    (fun y : tuple ⇒
     Oeset.nb_occ OTupleT y
       (Febag.elements BTupleT (eval_query_rel env (to_sql_query q)))) =
Oeset.nb_occ e
    (map
       (fun t ⇒
        projection (env_t env t) l)
       (Febag.elements (eval_query_rel env (to_sql_query q))))
```

Passer de la fonction de projection de la how-provenance à celle de Datacert se fait de façon similaire à l'aide du lemme de l'équivalence entre les deux fonctions :

```
Lemma projection_equiv : ∀ (env : env T) (l : list (select T)),
well_formed_aggregate l = true →
eq OTupleT
  (projection_prov
     (create_env env) (Select_List (_Select_List l)))
  (projection T env (Select_List (_Select_List l)))
```

Je ne m'étends pas ici sur la preuve de ce lemme. Il est à noter nous devons avoir la bonne formation de `l` pour que le lemme soit applicable, ce qui est assurée par la bonne formation de la requête.

La dernière étape est donc de passer du `support` aux éléments du multi-ensemble de `q`.

Pour faciliter la compréhension, je propose de s'intéresser à cette preuve en abstrayant certains éléments. Prouver l'égalité qui reste revient à prouver le lemme qui suit une fois qu'on a rentré la fonction de filtrage dans la fonction appliquée par `fold_left_wacc`.

```
Lemma fold_left_nb_occ : ∀ l s e f1,
    (∀ a b, Oeset.eq_bool a b = true →
          Oeset.eq_bool (f1 a) (f1 b) = true) →
    (∀ t, Oeset.mem_bool t l = true →
          Oeset.mem_bool OA t (Feset.elements s) = true) →
    fold_left_wacc N.add (Feset.elements s) 0
    (fun y ⇒
     (if Oeset.eq_bool e (f1 y) then Oeset.nb_occ y l else 0)) =
    Oeset.nb_occ e (map f1 l).
```

La difficulté ici est que `Feset.elements s` (en orange) donne une liste avec des éléments deux à deux distincts alors que `l` (en violet) peut avoir des éléments qui apparaissent plusieurs fois. Je pose pour la suite `l'` comme étant égale à `Feset.elements s`.

La première étape de cette preuve est d'éliminer les éléments de `l` et `l'` qui après application de `f1` ne sont pas équivalents à e car ceux-ci ne sont pas pris en compte dans les sommes. Cela permet d'éliminer dans le corps de `fold_left_wacc` la condition "if then else". De plus, le nombre d'occurrences de `e` dans `map f1 l` revient au nombre d'éléments restant de `l`.

Pour éliminer ces éléments de `l` (et réciproquement de `l'`), on divise la liste en deux sous-listes `l1` et `l2` (réciproquement `l1'` et `l2'`) avec `l1` contenant

les éléments de `l` vérifiant la condition et `l2` contenant les éléments restants de `l`. La concaténation des sous-listes `l1` et `l2` donne une permutation de `l`. Comme les fonctions `fold_left_wacc` et `Oeset.nb_occ` sont compatibles avec les permutations de listes, nous avons le nouveau but :

```
fold_left_wacc N.add (l1'++l2') 0
(fun y ⇒
 (if Oeset.eq_bool e (f1 y) then Oeset.nb_occ y (l1 ++l2) else 0)) =
Oeset.nb_occ e (map f1 (l1++l2)).
```

La condition "if then else" (en cyan) est simplifiable en seulement la partie du "then" car `l1'` ne contient que des éléments vérifiant la condition et chaque élément de `l2'` n'ajoute que des 0. Après simplification, nous obtenons le but :

```
fold_left_wacc N.add l1' 0
(fun y ⇒ Oeset.nb_occ y (l1 ++l2)) =
Oeset.nb_occ e (map f1 l1).
```

Le but peut être alors réécrit en remplaçant la liste `l1++l2` contenu dans la partie en cyan en `l1`. Les éléments de `l1'` vérifie la condition "`f1 y` est équivalent à `e`". Cependant, l'application de `f1` sur les éléments de `l2` ne peut pas être égale à `e`, ce qui rend impossible à `y` d'être équivalent à un élément de `l2`.

```
fold_left_wacc N.add l1' 0
(fun y ⇒ Oeset.nb_occ y l1) =
Oeset.nb_occ e (map f1 l1).
```

La deuxième partie de la preuve consiste à abstraire la liste `l1` puis à faire une induction sur la liste `l1'`. Cependant, comme précisé précédemment les éléments de `l1'` sont deux à deux distincts alors que les éléments de `l1` peuvent être redondants. Pour appliquer l'hypothèse d'induction, il est nécessaire de regrouper dans les listes les éléments équivalents.

Par exemple, prenons des éléments t_1, t_2 et t_3 non égaux entre eux et `l1` la liste `[t1;t3;t1;t1;t2;t3]`.
Avec les hypothèses du lemme, la liste `l1'` est égale à la liste `[t1;t2;t3]` à permutation près.

Lorsqu'on veut effectuer l'induction sur la queue de la liste `l1'= [t2;t3]`, on ne peux l'appliquer que pour la sous-liste de `l1`, `[t3;t2;t3]`. Il faut donc procéder comme précédemment : diviser la liste `l1` en deux sous-listes `l3` et `l4` tel que `l3` contient seulement des n-uplets équivalents à `t1` et `l4` prend les autres. Le reste de la preuve est assez directe.

On notera que pour obtenir la conclusion du lemme intermédiaire `fold_left_nb_occ`, il est seulement nécessaire d'avoir le lien entre les éléments des listes `l1` et `l1'` en hypothèse.

Fonctions d'agrégation

Le but à démontrer dans le cas des fonctions d'agrégation fait appel à de nombreuses fonctions intriquées les unes dans les autres, ce qui peut compliquer

la compréhension. Afin de faciliter celle-ci, je propose un exemple d'application des fonctions d'agrégation, qui servira d'illustration pour mieux appréhender le fonctionnement des différentes fonctions impliquées : La **Table 4.32** est le résultat

	Nom	**Age**	**Pays**
t_1	Martin	27	France
t_2	Durant	81	France
t_1	Martin	27	France
t_3	Martin	16	Belgique
t_4	Martin	92	Belgique
t_5	Chevalier	54	France
t_5	Chevalier	54	France

Table 4.31 – relation r_{15}

	Nom	**C**
t_6	Martin	2
t_6	Martin	2
t_7	Durant	1
t_8	Chevalier	2

Table 4.32 – $Nom := Nom, C := Compte(Age)$ groupé par nom et pays

de l'application de la fonction d'agrégation "Compte" sur l'attribut "Age" après le groupement par nom et pays. On affiche aussi la colonne "Nom". Je souhaite attirer l'attention ici sur l'ordre des n-uplets : les n-uplets sont triés par ordre alphabétique sur l'attribut "Nom", puis s'il y a égalité entre deux par ordre croissant sur "Age" et s'il y égalité par l'ordre alphabétique sur "Pays". La représentation graphique faite ici des n-uplets ne nécessite pas de respecter l'ordre, mais celui-ci aura de l'importance dans les résultats renvoyés par les fonctions utilisées.

Pour prouver l'égalité entre les annotations d'un n-uplet et son nombre d'occurrences dans le cas des fonctions d'agrégation, nous devons prouver :

```
f_gamma
    (map
       (fun l0 ⇒
        (projection_prov
           (env_gp (create_env env) (Group_By f0) l0) l), l0))
       (make_groups_prov
          (create_env env) (eval_prov_N (create_env env) q)
          (Group_By f0))) e =
  Febag.nb_occ e
    (Febag.mk_bag
       (map
          (fun l0 ⇒
           projection (env_g env (Group_By f0) l0) l)
          (make_groups env
             (Febag.elements (eval_query_rel env (to_sql_query q)))
```

(Group_By f0))))

La partie de l'égalité héritée de Datacert est en cyan et celle de la how-provenance est en violet

Application à l'exemple

Regardons de plus près ce qu'il se passe dans le cas où notre sous-requête est la **Table 4.31** sur laquelle on applique la fonction d'agrégation proposée ci-avant. `make_groups_prov` donne la liste des K-relations des différents groupes créés à partir de la fonction de groupement `f0` qui est ici l'attribut "Nom" et l'attribut "Pays". Elle renvoie la liste : $l_k^1 = [k_4; k_2; k_3; k_1]$ où

$$k_1 = x \rightarrow \begin{cases} 2 \text{ si } x = t_1 \\ 0 \text{ sinon} \end{cases} \qquad k_3 = x \rightarrow \begin{cases} 1 \text{ si } x = t_3 \\ 1 \text{ si } x = t_4 \\ 0 \text{ sinon} \end{cases}$$

$$k_2 = x \rightarrow \begin{cases} 1 \text{ si } x = t_2 \\ 0 \text{ sinon} \end{cases} \qquad k_4 = x \rightarrow \begin{cases} 2 \text{ si } x = t_5 \\ 0 \text{ sinon} \end{cases}$$

k_1 correspond au groupement nom-pays ("Martin", "France"), k_2 pour ("Durant", "France"), k_3 pour ("Martin", "Belgique") et k_4 pour ("Chevalier", "France"). On remarque ici que l'ordre dans lequel apparaît les groupes dans la liste l_k^1 dépend de l'ordre des n-uplets. L'application de la fonction `projection_prov` et de l'identité sur les différents éléments de la liste donne $l_k^2 = [(t_8, k_4); (t_7, k_2); (t_6, k_3); (t_6, k_1)]$. La fonction `f_gamma` applique la fonction δ sur les n-uplets résultants et donne la K-relation suivante :

$$k_{res} = x \rightarrow \begin{cases} 1 \text{ si } x = t_8 \\ 1 \text{ si } x = t_7 \\ 2 \text{ si } x = t_6 \\ 0 \text{ sinon} \end{cases}$$

Pour la partie venant de Datacert, `make_groups` donne la liste des listes de n-uplets des différents groupes : $l_b^1 = [[t_5; t_5]; [t_2]; [t_3; t_4]; [t_1, t_1]]$. La première liste correspond au groupement nom-pays ("Chevalier", "France"), la seconde pour ("Durant", "France"), la troisième pour ("Martin", "Belgique") et la quatrième pour ("Martin", "France"). L'application de la fonction `projection` sur les différents éléments de la liste donne $l_b^2 = [t_8; t_7; t_6; t_6]$. La fonction `Febag.mk_bag` convertit la liste en multi-ensemble.

Preuve

Maintenant que nous avons vu l'application des fonctions principales sur l'exemple, nous passons à la preuve. Dans un premier temps, j'ai choisi de réécrire l'égalité pour que chaque partie de celle-ci utilise la même fonction comptant le nombre d'occurrences de la bibliothèque `Oeset`, c'est-à-dire d'avoir une égalité de la forme :

```
Oeset.nb_occ e l1 = Oeset.nb_occ e l2
```

où la fonction `Oeset.nb_occ` donne le nombre de fois où e apparaît (à équivalence près) dans la liste `l1` ou `l2`. Si on reprend l'exemple, les listes `l1` et `l2` correspondent aux listes l_k^2 et l_b^2.

Pour la partie de l'égalité héritée de Datacert (en cyan), il suffit donc de passer du multi-ensemble généré par `Febag.mk_bag` à la liste des éléments de ce multi-ensemble. On passe de `Febag.nb_occ e (Febag.mk_bag l2)` à `Oeset.nb_occ e l2`.

Pour ce qui est de la fonction `f_gamma` (en violet), nous avons le lemme suivant :

```
Lemma f_gamma_nb_occ : ∀ l t,
    (∀ x, In x l → delta_it_f (snd x) = 1) →
    f_gamma l t =
    Oeset.nb_occ t (map fst l).
```

La liste `l` est une liste de couples n-uplet-K-relation. Si on se réfère à l'exemple, elle correspond à la liste l_k^2. Le lemme `f_gamma_nb_occ` indique que si pour chaque couple (t_k, k) de la liste `l`, k a au moins un n-uplet avec une annotation non nulle, la fonction `f_gamma` pour un n-uplet `t` est égale au nombre de fois où `t` est équivalent au premier élément de la liste `l`. L'hypothèse dans notre cas est vérifiée par définition de la fonction `make_groups_prov`.

Après application du lemme et quelques simplifications, le but à prouver devient le suivant :

```
Oeset.nb_occ e
  (map
     (fun x : ⇒
      projection_prov (env_gp (create_env env) (Group_By f0) x) l)
     (make_groups_prov (create_env env) (eval_prov_N (create_env env) q)
        (Group_By f0))) =
Oeset.nb_occ e
  (map
     (fun l0 ⇒
      projection (env_g T env (Group_By f0) l0) l)
     (make_groups T env
        (Febag.elements (eval_query_rel env (to_sql_query q)))
        (Group_By f0)))
```

Il est à noté ici que la fonction (en violet) n'est pas exactement la même que précédemment. On a éliminé le deuxième élément de la paire. Par rapport à l'exemple, cela signifie que la liste qui est renvoyée est maintenant $l_k^3 = [t_8; t_7; t_6; t_6]$.

Il faut maintenant montrer que le n-uplet e apparaît autant de fois dans les deux listes. Avec l'exemple, nous voyons que les listes l_k^3 et l_b^2 sont égales et cela nous donne l'intuition que les deux listes sont les mêmes à équivalence près des n-uplets. On utilise alors un lemme qui permet de passer du but précédent avec une égalité entre le nombre d'occurrences de deux listes à un but de la forme :

```
Forall2
 (fun a1 a2 ⇒ Oeset.eq_bool a1 a2 = true)
  l1 l2
```

où `Forall2` prend les éléments des listes `l1` et `l2` et regarde si pris deux à deux, ils vérifient la relation `(fun a1 a2 => Oeset.eq_bool a1 a2 = true)`. Ici `l1` et `l2` correspondent dans l'exemple aux listes l_k^3 et l_b^2.

Vu que les fonctions `projection_prov` en violet et `projection` en orange sont appliquées à chaque éléments des listes (en cyan), nous pouvons aller encore plus loin et modifier le but de façon à ce que pour chaque élément des listes en violet pris deux à deux vérifie la relation :

```
fun a1 a2 ⇒ Oeset.eq_bool
   (projection_prov (env_gp (create_env env) (Group_By f0) a1) l)
   (projection (env_g env (Group_By f0) a2) l) = true
```

Pour vérifier cette condition, il est nécessaire de réécrire soit `projection` en `projection_prov` ou inversement. Avec le lemme `projection_equiv` vu dans le cas précédent, on a pour un élément a2 que :

```
projection (env_g T env (Group_By f0) a2) l
```

est équivalent à

```
projection_prov (env_gp (create_env env) (Group_By f0) a2) l
```

Nous savons aussi avec le lemme `projection_prov_eq` que si deux environnements sont équivalents alors les résultats de la fonction `projection_prov` appliquée sur ces deux environnements (avec la même fonction de groupement) sont équivalentes.

La condition devient alors :

```
fun a1 a2 ⇒ Oeset.eq_bool
    equiv_envp (env_gp (create_env env) (Group_By f0) a1)
               (create_env (env_g env (Group_By f0) a2)).
```

Le but devient alors le suivant (après déroulage des fonctions `make_groups_prov` et `make_groups`) :

```
Forall2
 (fun (x : list value * Krel) (y : list value * listT) ⇒
       equiv_envp (env_gp (create_env env) (Group_By f0) (snd x))
                  (create_env (env_g env (Group_By f0) (snd y))))
 (partition_rec_krel
    (fun t ⇒
     map
       (fun h ⇒
        interp_aggterm_prov
```

```
            (env_gp (create_env env) (Group_By f0)
                {|
                f := fun t0 : tuple ⇒
                      if Oeset.eq_bool OTupleT t0 t then 1 else 0;
                support := Feset.singleton FesetT t;
                sort_krel := sort_krel kr |}) h) f0) kr nil
    (Feset.elements FesetT (filter_zero_N kr)))
  (partition_rec
    (fun t ⇒
     map
       (fun f1 : aggterm ⇒
        interp_aggterm (env_g T env (Group_By f0) (t ; nil)) f1) f0) nil
    (Febag.elements BTupleT fb))
```

La différence entre les listes renvoyées par les fonctions `partition` et par les fonctions `make_groups` ici est que les premières indiquent pour chaque groupe quelles sont les valeurs prises par ce groupe pour les expressions d'attributs du groupement alors que les secondes ne donnent que les éléments des groupes. En effet, les éléments des listes renvoyées par les fonctions `partition` sont des couples dont le premier élément correspond aux valeurs prises par le groupe constituant le second élément du couple.
En d'autres termes, en reprenant notre exemple, nous avons :

— au lieu de la liste l_k^1, la liste :
$l_k^{par} = [([\text{Chevalier}; \text{France}], k_4); ([\text{Durant}; \text{France}], k_2); ([\text{Martin}; \text{Belgique}], k_3); ([\text{Martin}; \text{France}], k_1)]$

— au lieu de la liste l_b^1, la liste :
$l_b^{par} = [([\text{Chevalier}; \text{France}], [t_5; t_5]); ([\text{Durant}; \text{France}], [t_2]); ([\text{Martin}; \text{Belgique}], [t_3; t_4]); ([\text{Martin}; \text{France}], [t_1, t_1])]$

Il est plus simple d'effectuer la preuve au niveau des fonctions `partition` qu'au niveau des fonctions `make_groups`. Connaître les valeurs prise par chaque groupe sur la fonction de groupage permet de savoir à quel groupe un élément est ajouté lors de l'application des fonctions `partition`. Sans cela, déterminer le groupe en question devient laborieux et complique la preuve.

Nous notons par ailleurs qu'en prenant seulement les premiers éléments de chaque couple de l_k^{par} et l_b^{par}, nous avons la même liste : les groupes sont donc bien renvoyés dans le même ordre. Comme cette information est primordiale pour la preuve, et confirme que chaque K-relation et liste de n-uplets pris deux à deux désigne bien les mêmes groupes.

Je renforce ici la condition que doit vérifier les deux listes colorées en cyan par la condition que chaque couple d'éléments des listes pris deux à deux doit avoir le même premier élément (donc correspondre au même groupe).
La nouvelle condition est la suivante :

```
Definition R_val_krel env f0 l := (fun (x : list value * Krel N) (y : list value * listT) ⇒
    fst x = fst y /\
    equiv_envp (env_gp (create_env env) (Group_By f0) (snd x))
               (create_env (env_g env (Group_By f0) (snd y)))).
```

Pour montrer que cette condition est vérifiée par mes listes, j'ai proposé de découper la preuve en trois sous-preuve grâce au lemme suivant :

```
Lemma projection_Forall2_equiv : ∀ l1 l2 s env f0,
    Forall (fun (x : list value * Krel N) ⇒
              s ≡sort_krel (snd x)) l1 →
    Forall (R_val_krel3 s) l2 →
    Forall2 R_val_krel2 l1 l2 →
    Forall2 R_val_krel l1 l2.
```

où les relations sont :

```
Definition R_val_krel2 := (fun (x : list value * Krel N) (y : list value * listT) ⇒
     fst x = fst y ⋀
     filter_zero_N (snd x) =SE= Feset.mk_set FesetT (snd y) ⋀
     (∀ t : tuple, f (snd x) t = Oeset.nb_occ OTupleT t (snd y))).


Definition R_val_krel3 s := (fun (y : list value * listT) ⇒
     (∃ t : tuple, Oeset.mem_bool OTupleT t (snd y) = true) ⋀
     (∀ t : tuple,
      Oeset.mem_bool OTupleT t (snd y) = true →
      s ≡labels T t)).
```

En reprenant la définition d'une équivalence entre deux environnements, les relations `R_val_krel2` et `R_val_krel3` prennent leur sens.

Les deux conditions de `R_val_krel2` correspondent à deux des conditions sur les éléments de deux environnements équivalents. Il s'agit de la condition sur les `supports` et sur les fonctions d'annotations des éléments pris deux à deux dans les environnements. Les deux conditions ont été un peu réécrites et simplifiées grâce à la définition de la `create_env`.

Il reste maintenant la condition sur les `sortes` des éléments de deux environnements équivalents. `R_val_krel3` assure que les n-uplets doivent tous avoir la même sorte `s` pour la liste `l2`. On retrouve cette même condition pour `l1`.

Nous observons que dans `R_val_krel3`, nous avons une condition supplémentaire qui est de vérifier qu'il existe au moins un élément dans les groupes de la liste provenant de la sémantique de Datacert. Cette propriété avait déjà été abordée pour les groupes issus de la how-provenance et est donc nécessaire pour les deux listes.

Maintenant que la preuve est découpée en sous-preuves, vient la deuxième étape délicate. Chacune de ces sous-preuves nécessite d'abstraire la liste `Febag.elements BTupleT fb` puis faire une induction sur la liste `Feset.elements (filter_zero_N kr)`. De plus, les éléments peuvent apparaître plusieurs fois pour `Febag.elements`. Nous nous retrouvons dans un cas similaire au cas de la projection : nous avions pu découper les listes `Feset.elements s` et `l` et faire l'induction sans se préoccuper de l'ordre des éléments. La différence dans notre cas est qu'ici l'ordre des éléments joue un rôle majeur : nos conditions

reposent sur le fait que les éléments des listes correspondent au même groupe deux à deux.

Nous prenons donc en compte que les fonctions `Feset.elements` et `Febag.elements` renvoie des listes triées par ordre croissant selon la relation d'ordre.

Reprenons l'exemple proposé dans le cas de la projection. Nous avions une table avec les n-uplets t_1 trois fois, t_2 une fois et t_3 deux fois. Nous ajoutons les relations entre les n-uplets : $t_1 < t_2 < t_3$. On note `fb`, le multi-ensemble correspondant et `kr` la relation associé.

Cette fois-ci, on a les n-uplets triés dans l'ordre pour les deux listes.
`Febag.elements BTupleT fb` donne la liste `[t1;t1;t1;t2;t2;t3]`.
`Feset.elements BTupleT (filter_zero_N kr)` donne la liste `[t1;t2;t3]`.

Ici lorsqu'on divise les listes selon que les éléments sont équivalents ou non à `t1`, nous pouvons le faire pour que les sous-listes obtenues soient égales et non plus seulement une permutation des listes initiales grâce au fait que les listes sont triées. Lorsqu'on veut effectuer l'induction sur la liste de la K-relation pour prouver les propriétés de `R_val_krel2`, on va ainsi l'appliquer sur les listes `[t2;t2;t3]` (multi-ensemble) et `[t2;t3]` (K-relation).

Les sous-lemmes requièrent aussi d'abstraire les accumulateurs : ceux-ci doivent alors de vérifier les hypothèses de `projection_Forall2_equiv`. Le reste des preuves ne porte pas de difficulté supplémentaire.

Cette preuve confirme ainsi que la sémantique de notre modèle coïncide avec la sémantique de la bibliothèque de Datacert sans annotation. Ce théorème est aussi vérifié pour la version de la sémantique pour les annotations sur l'algèbre relationnelle positive.

4.7 . Instanciation

La deuxième instanciation se concentre sur la proposition d'une version concrète et exécutable de la how-provenance. Elle vise à mettre en évidence la structure de semi-module sous-jacente aux domaines des valeurs sur lesquelles sont appliquées des fonctions d'agrégation.

Cette section se décompose en trois sous-niveaux d'instanciation. Le premier reprend l'instanciation des fonctions proposées dans Datacert à l'exception des fonctions d'interprétation des fonctions d'agrégation. Ces dernières ne seront que partiellement instanciées pour mettre en évidence la structure de K-semi-module. Le semi-anneau commutatif reste donc abstrait à ce niveau.
Le deuxième niveau instancie, en complément, le semi-anneau avec l'ensemble des entiers naturels.
Dans un dernier temps, les fonctions d'interprétation des fonctions d'agrégation sont instanciées par les fonctions "Minimum", "Maximum", "Compte" et "Som-

me". Il est à signaler que les deux dernières instanciations vérifient le théorème d'adéquation (**Section 4.6**). Afin de faciliter la comparaison entre Datacert et la how-provenance, nous mettons en lumière les liens entre les fonctions d'interprétation des fonctions d'agrégation de Datacert et celles qui prennent en compte les annotations à chaque étape.

Dans une volonté de proposer une version concrète et exécutable complète de la sémantique de la how-provenance, nous devons proposer une structure pour les bases de données, instancier les structures de données (types et fonctions de l'enregistrement `Tuple`) et le semi-anneau commutatif.

Niveau 1

Dans cette première instanciation, le semi-anneau est laissé abstrait. Il est noté ici (`K, zero, one, add, mul, TK`).

Bases de données

Comme nous souhaitons comparer la sémantique concrète de Datacert avec celle de cette formalisation, je définis les bases de données sources dans le modèle multi-ensembliste :

```
Record db_state_ (T : Rcd) : Type := mk_state
  { _relnames : list relname;
    _basesort : relname → Fset.set (A T);
    _instance : relname → Febag.bag (Fecol.CBag (CTuple T)) }
```

Le premier champ donne les noms des tables sources. On note ici que `relname` est le type instancié des noms des tables sources : le nom est donné sous la forme `Rel chr` où `chr` est une chaîne de caractère. Le deuxième champ indique pour chaque table source sa sorte. Le dernier champ donne pour chaque table source le multi-ensemble des n-uplets qui la constituent.

À partir d'une base de données source `db`, il est possible d'en déduire (en partie) les champs des K-relations sources associées. Les champs `support` et `sort_krel` d'une table source de `db` découlent réciproquement du champ `_instance` et du champ `_basesort`. Cependant, le semi-anneau étant abstrait, il n'y a pas de lien direct entre les annotations et le nombre d'occurrences des n-uplets. Il est ainsi nécessaire de poser en hypothèse une fonction qui à chaque table source d'une base de données associe sa fonction des annotations :

```
Hypothesis f_instRA : db_state_ TNullRA → relname → Tuple.tuple TNullRA → K.
```

Structure de données

Il apparaît que hormis la fonction d'interprétation des fonctions d'agrégations (`interp_aggregate_prov`), les fonctions et types des structures de données (enregistrement `Tuple`) utilisées pour la sémantique de la how-provenance ne dépendent

pas des annotations des n-uplets et sont donc instanciées de façon similaire à celles de Datacert.

La fonction d'interprétation des fonctions d'agrégation nécessite un effort supplémentaire : les valeurs sur lesquelles les fonctions d'agrégation sont appliquées sont plongées dans un semi-module M et les valeurs renvoyées sont le résultat d'opérations de M et du semi-anneau `K`. Pour faire ressortir cette structure de semi-module, les fonctions d'interprétation d'agrégation sont partiellement définies.

Fonctions d'interprétation

Je rappelle que, dans la formalisation abstraite, la fonction `interp_aggregate_prov` est laissée en hypothèse et attend une fonction d'agrégation et une liste de couples $(valeur * annotation)$ et applique la fonction sur la liste.

Dans le but de simplifier le modèle, l'instanciation de la fonction d'interprétation est divisée en deux étapes : une étape qui associe (`interp_aggregate_abs_prov`) les valeurs de la liste à des valeurs dans les entiers relatifs et une étape (`interp_aggregate_abs_provZ`) qui effectue l'interprétation de la fonction d'agrégation sur les nouvelles valeurs. Pour faciliter la comparaison, la fonction d'interprétation des fonctions d'agrégation sans annotations est elle aussi divisés en deux étapes (étape 1 :`interp_aggregate_abs`, étape 2 : `interp_aggregate_absZ`).

Étape 1

Bien qu'une valeur dans notre formalisation puisse être soit une chaîne de caractères, soit un entier relatif, soit un booléen, soit la valeur NULL, nous pouvons raisonner sur des valeurs de type `option Z` lorsqu'on applique les fonctions d'agrégation. En effet, trois des fonctions d'agrégation dans notre modèle peuvent exclusivement s'appliquer sur des entiers relatifs et la valeur Null. Seule la fonction "Compte" peut s'appliquer sur les autres domaines. Dans ce cas précis, il suffit d'associer tous les éléments d'un des domaines à l'entier 1 et pour les valeurs Null associer l'entier 0 et nous obtenons le même résultat que l'on soit dans l'un ou l'autre des domaines. Une liste `[(''a'',1);(''i'',3);(Null,2);(''u'',7)]` où "a" apparaît 1 fois, "i" 3 fois et "u" 2 fois, devient la liste `[(1,1);(1,3);(0,2);(1,7)]`. Si on applique la fonction "Compte" sur l'une des deux listes, elle donne le même résultat 11.

Il convient de souligner que dans le travail [10], l'étape 1 est faite vers l'ensemble des réels. Cependant, avec le type des domaines proposés dans l'instanciation, nous ne perdons pas d'information en nous restreignant à l'ensemble des entiers relatifs. De plus, on note aussi que certaines fonctions d'agrégation usuelles

ne correspondent au modèle avec les semi-modules. C'est le cas de la fonction qui fait la moyenne d'un groupe de valeur [10]. Nous posons donc ici l'hypothèse `Hypothesis wf_aggregate : aggregate -> bool` qui vérifie que le nom de la fonction d'agrégation ne fait pas partie de ces cas non-désirés.

Étape 2

La fonction abstraite `interp_aggregate_abs_provZ` correspond à la deuxième étape de la fonction d'interprétation, et c'est à ce stade que la structure de semi-module devient visible.

```
Hypothesis interp_aggregate_abs_provZ : aggregate → list (option Z * K) → option Z.
```

Elle prend en argument le nom de la fonction d'agrégation à appliquer, une liste des valeurs dans `option Z` et leur annotations associées et renvoie le résultat de l'application de la fonction d'agrégation sur la liste.

Même si l'ensemble dans lequel les valeurs sont plongées est toujours le type `option Z`, les autres éléments du monoïde dépendent de la fonction d'agrégation appliquée. Le "zéro" correspond à la valeur de `interp_aggregate_abs_provZ` sur la liste vide. L'opération du monoïde $\oplus$ est laissée abstraite à ce stade :

```
Hypothesis plus_valuesZ : aggregate → option Z → option Z → option Z.
```

L'opération $\oplus$ s'applique entre deux résultats de l'application de la fonction d'agrégation. Nous posons donc l'hypothèse `interp_aggregate_abs_provZ_app` : le résultat d'une concaténation de deux listes `l1` et `l2` par la fonction `interp_aggregate_abs_provZ` pour une fonction d'agrégation `a` est égale à l'application de `plus_valuesZ a` sur les résultats de `interp_aggregate_abs_provZ a` sur ces deux listes individuellement.

```
Hypothesis interp_aggregate_abs_provZ_app : ∀ a l1 l2,
    interp_aggregate_abs_provZ a (l1++l2) =
    plus_valuesZ a (interp_aggregate_abs_provZ a l1)
                   (interp_aggregate_abs_provZ a l2).
```

Pour ce qui est de l'opération $\otimes$, elle est elle aussi dépendante de la fonction d'agrégation et prend en argument un élément du semi-anneau `K` et une valeur.

```
Hypothesis tensorZ : aggregate → K → option Z → option Z.
```

Les caractéristiques que doit vérifier le tenseur $\otimes$ (présentées dans **Section 4.4**) ont été prises en compte dans nos hypothèses. Nous avons par exemple l'égalité $0_k *_K o = 0_K$ avec l'hypothèse :

```
Hypothesis tensorZ_eq_0 : ∀ a o, tensorZ a zero o = zero_Z a.
```

De plus, nous avons une hypothèse qui fait coïncider la fonction d'interprétation et le tenseur $\otimes$, lorsque la liste prise en entrée par la fonction d'interprétation ne contient qu'un seul élément :

```
Hypothesis tensorZ_eq_1 : ∀ a k z,
    tensorZ a k z = interp_aggregate_abs_provZ a ((z,k) :: nil).
```

Le K-semi-module est conditionné par le choix de la fonction d'agrégation (qui doit satisfaire la condition `wf_aggregate`). La preuve est laissée abstraite à ce stade :

```
Hypothesis M_K_SemiModule : ∀ a,
    wf_aggregate a = true →
    SemiModule K (option Z) zero one add mul (zero_Z a) (plus_valuesZ a)
          (tensorZ a) TK OeValZ.
```

Il est important de noter que les fonctions d'interprétation de Datacert (`interp_aggregate_absZ`) et de notre formalisation (`interp_aggregate_abs_provZ`) doivent coïncider dans certains cas. Par exemple, pour une liste vide, les deux fonctions doivent être égales si la fonction d'agrégation est la même. Cette valeur renvoyée correspond au "zéro" du monoïde commutatif.

À ce niveau d'abstraction, il est intéressant de noter que l'une des hypothèses additionnelles (`interp_aggregate_prov_prop1`) nécessaire pour avoir le théorème d'adéquation (voir section précédente) est vérifiée. Elle ne dépend pas du semi-anneau choisi et nécessite seulement d'avoir une structure de semi-module sur les valeurs. Ce n'est cependant pas le cas de la seconde hypothèse `interp_aggregate_prov_prop2`.

Niveau 2

La deuxième instanciation va plus loin en instanciant également le semi-anneau commutatif par l'ensemble des entiers naturels : on reprend ainsi certains éléments de la **Section 4.6** comme la définition de δ_N.

A ce niveau d'instanciation, nous souhaitons avoir le théorème d'adéquation. Pour ce faire, nous faisons coïncider le nombre d'occurrences d'un n-uplet des tables sources avec les annotations que prend ce n-uplet. Les fonctions des annotations des tables sources sont ainsi définies à partir du champ `instance_rel_` d'une base de données :

```
Definition f_inst db r t := Febag.nb_occ t (instance_rel_ db r).
```

Pour que le théorème d'adéquation soit vérifié, il faut que les K-relations sources `r` d'une base de données `db` vérifient les invariants de `Krel_inv`. Pour cela, j'impose la condition suivante sur les bases de données sources :

```
Definition well_formed_db db r :=
Fset.for_all (fun x ⇒ Fset.equal (labels x) (_basesort db r)) (support_inst db r).
```

Celle-ci force la sorte des n-uplets d'une table source à correspondre à la sorte de cette table. Cette condition est suffisante pour que `Krel_inv` soit vérifiée.

Afin de démontrer la deuxième hypothèse supplémentaire `interp_aggregate_prov_prop2` de la **Section 4.6**, l'utilisation d'une fonction intermédiaire est requise :

```
Fixpoint decompose_list l :=
  match l with
  |nil ⇒ nil
  |(x,y) :: tl ⇒ if Oeset.eq_bool y 0 then decompose_list tl
                 else (repeat (x, 1) y) ++decompose_list tl
  end.

Lemma decompose_interp_aggregate_abs : ∀ a l,
    wf_aggregate a = true →
     interp_aggregate_abs_prov a l = interp_aggregate_abs_prov a (decompose_list l).
```

Avec le lemme `decompose_interp_aggregate_abs`, la fonction `decompose_list` réduit la preuve de `interp_aggregate_prov_prop2` des listes quelconques aux listes contenant seulement des couples de la forme (x,1). Avec les hypothèses de `interp_aggregate_prov_prop2`, cela implique que les listes sont une permutation l'une de l'autre. Suite à cela, il est relativement aisé de prouver ce lemme.

Avec cette preuve, on retrouve le théorème d'adéquation si les conditions `well_formed_db` et `wf_aggregate` sont vérifiées :

```
Lemma K_relations_extend_relational_algebra_inst_abs : ∀ b db,
    (∀  r, well_formed_db db r = true) →
    ∀ env q,
      well_formed wf_aggregate env b q = true →
      ∀ t,
        f (eval_prov_Null2_N db (create_env env) q) t =
        Febag.nb_occ t (eval_query_rel_Null2 db env q).
```

Niveau 3

La dernière instanciation consiste à instancier la fonction `interp_aggregate_abs_provZ` avec les fonctions d'agrégation "Compte", "Somme", "Minimum" et "Maximum".

L'implémentation des fonctions d'agrégation ne pose pas de difficultés particulières. Cependant, il est important de noter que la valeur NULL est ignorée lors du calcul de ces fonctions. Les fonctions d'agrégation ne renvoient une valeur NULL que si la liste des valeurs est une liste constituée uniquement de valeurs NULL ou une liste vide. Il en est de même pour la fonction `plus_valuesZ`.

Comme vu précédemment la valeur renvoyée par le tenseur $\otimes$ sur une valeur `v` et une annotation `a` doit être la même que si la fonction `interp_aggrefate_abs_provZ` est appliqué sur la liste `[(v,a)]`. Si la valeur est None, la valeur None est renvoyée. Dans le cas où la valeur n'est pas Null, si l'annotation est égale à 0, alors la valeur renvoyée est Null. Sinon :

- l'application de la fonction "Minimum" sur `v` n'est pas affectée par l'annotation et renvoie `v`
- il en est de même pour "Maximum"
- comme la valeur `v` apparaît `a` fois dans la table, l'application de "Somme" sur `[(v,a)]` renvoie la multiplication de `v` et `a`

— pour la fonction "Compte", v va prendre soit la valeur 0 (pour les valeurs qui étaient Null avant l'étape 1) soit la valeur 1 (pour les autres valeurs). Il ne reste donc plus qu'à faire la multiplication entre v et a pour obtenir le nombre de fois où la valeur apparaît.

Grâce à cette instanciation, nous pouvons constater que pour chacune des quatre fonctions d'agrégation, l'ensemble `option Z` est un `N`-semi-module et nous en déduisons le théorème d'adéquation.

On notera que la version de la sémantique restreinte sur l'algèbre relationnelle positive a une instanciation concrète similaire à celle proposée. De plus, il est intéressant de remarquer ici que les 3 instanciations présentées prennent en compte la valeur spéciale `NULL` dans les différentes fonctions utilisées.

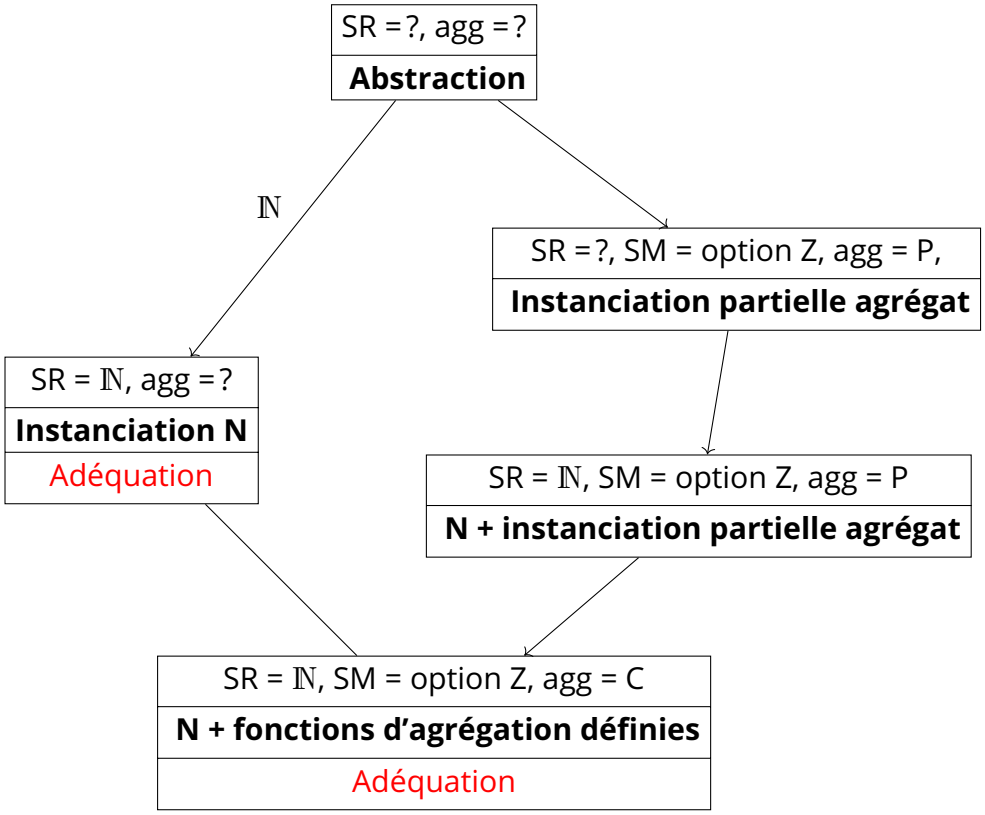


Figure 4.1 – Instanciation de l'évaluation des requêtes

Le **Schéma 4.1** reprend les instanciations qui ont été proposées. À chaque étape, il est précisé l'instanciation du semi-anneau (? = pas instancié, N = entiers naturels), l'instanciation du semi-module et l'état de l'instanciation des fonctions d'interprétation des fonctions d'agrégation (P = instanciation partielle, C = instanciation complète).

4.8 . Exemple d'application de la how-provenance

Nous proposons dans cette section d'appliquer la sémantique concrète sur deux exemples de requêtes. Pour plus de lisibilité, nous utilisons le parseur et compilateur de la bibliothèque Datacert qui permet de passer d'une requête écrite dans la syntaxe de SQL vers une requête écrite dans celle de l'algèbre relationnelle positive étendue.

Nous pouvons par exemple écrire la commande suivante :
`Parse_sql "select * from table1;" q1.` qui associe à `q1` la requête sélectionnant tous les éléments de la table source nommée `table1`.

Nous partons de trois tables :

Name	Rem	Tr	S
Smith	true	A	1
Jones	false	B	2
Garcia	false	A	4
Smith	true	A	2

Table 4.33 – table1

Name	Rem	Tr	S
Miller	true	B	2
Jones	false	B	2
Johnson	true	A	2

Table 4.34 – table2

Name	City
Smith	Cambridge

Table 4.35 – table3

Je propose de m'intéresser à quelques évaluations de la fonction `eval_prov` pour les requêtes qui suivent.

```
Parse_sql "select count(name) as c from (table table1 union
table table2) where (stage <= 2 and remission = true) group by
treatment;" q2.
```

fait l'union des deux tables (`table1` et `table2`) puis ne conserve que les n-uplets dont le stade d'avancée de la maladie est inférieur ou égal à deux et où le patient est en rémission. Enfin, la requête applique la fonction "Compte" sur l'attribut "Name" après un groupement sur "Treatement". La colonne résultante est nommée "c".

Le résultat renvoyé par l'évaluation de la requête est :

```
(1, (AZ "c", Z 1) :: nil) :: (1, (AZ "c", Z 3) :: nil) :: nil
```

Cela se lit de la manière suivante :

- pour le n-uplet dont la valeur associée à l'attribut "c" de domaine dans `option Z` (élément en violet) est "1" (élément en cyan), l'annotation est égale à 1 (élément en orange)
- pour le n-uplet dont la valeur associée à l'attribut "c" de domaine dans `option Z` est "3", l'annotation est égale à 1

— le reste des n-uplets sont annotés à 0.

Cela correspond bien à ce qu'on attendait. En effet, nous avons 3 personnes dans l'union des deux tables qui sont en rémission, à un stade en deçà de 2 et qui prennent le traitement A et 1 seule pour le traitement B.

`Parse_sql "select name, city from table1, table3 where table1.name = table3.name;" q3.`
fait la jointure entre la `table1` et la `table3` puis projette sur "Name" et "City". On attend ainsi une K-relation qui est nulle partout sauf pour le n-uplet $(Smith, Cambridge)$ avec une annotation à 2 ce qui est confirmé par le résultat :

```
(2, (S "name", Value_string (Some "Smith"))
    :: (S "city", Value_string (Some "Cambridge"))
    :: nil) :: nil
```

Ici les noms de l'attribut ont des domaines de type `string` ce qui est désigné par `S` avant le nom de l'attribut mais aussi par `Value_string`.

4.9 . Discussion

Ce chapitre a proposé la formalisation abstraite de la how-provenance pour les requêtes de l'algèbre relationnelle étendue aux expressions d'attributs et aux fonctions d'agrégation, ainsi que plusieurs instanciations (semi-anneau des entiers, structures de données concrètes). Cette formalisation a été validée par le théorème d'adéquation montrant l'équivalence entre les sémantiques des requêtes sans annotation et avec annotations dans l'ensemble des entiers. Dans cette section, je propose de m'attarder sur les perspectives directes de travail de cette première contribution. Les perspectives à plus long terme seront discutées dans le **Chapitre 7**.

4.9.1 . Ajout d'expressivité

Une suite logique serait d'augmenter l'expressivité de la formalisation de la K-algèbre : permettre d'appliquer la formalisation aux requêtes comprenant des HAVING. Lors de l'application de fonctions d'agrégation sur des groupements de données, la composante HAVING permet de filtrer ces groupes selon certaines propriétés. Cette étape permettrait d'interroger des bases de données avec des requêtes "SELECT FROM WHERE GROUP BY HAVING". Pour rendre cela possible, il est nécessaire de changer la structure mathématique des annotations. À chaque n-uplet est associée une annotation plongée dans un semi-anneau commutatif, mais sont aussi associées des équations qui doivent être satisfaites par les annotations [10].

4.9.2 . Évaluation des performances et comparaison à d'autres systèmes

Une autre piste est de procéder à l'extraction en OCaml des bibliothèques Coq pour obtenir un code formellement vérifié.

L'obtention de ce code permettra de

— tester ce dernier avec des jeux de requêtes et de bases de données pouvant être supportés par notre expressivité, dont des jeux issus d'analyses bioinformatiques

— comparer les résultats de notre système avec d'autres systèmes d'annotations de données pour détecter des erreurs ou bugs présents dans ces derniers.

Un système que nous avons déjà en tête est l'extension PROVSQL de PostgreSQL [32], qui ajoute la possibilité d'annoter les bases de données et qui a une expressivité similaire à la nôtre.

4.9.3 . Deuxième validation du modèle : annotations booléennes

Sur le plan de la validité du modèle proposé, nous pouvons effectuer une preuve similaire au théorème d'adéquation mais avec comme semi-anneau les booléens : les n-uplets dont l'annotation est la valeur de vérité "vrai" devraient être dans le multi-ensemble issu des requêtes de Datacert et réciproquement. Cela permettrait de valider que la sémantique ensembliste est bien représentable par les annotations booléennes.

4.9.4 . Polynômes

Une dernière direction serait de faire la preuve que les polynômes sont bien un modèle générique des semi-anneaux, et qu'instancier les indéterminées avant ou après l'application des requêtes donne les mêmes annotations pour les n-uplets résultants.

4.10 . Différence entre les provenances

De nombreux travaux de la communauté des bases de données se sont intéressés à comprendre et mieux articuler les différences entre les sémantiques des divers types de provenance. La communauté des bases de données s'est notamment interrogée sur la possibilité qu'un semi-anneau puisse capturer la sémantique de la why-provenance ou celle de la where-provenance [33] sur l'algèbre relationnelle positive.

Why-provenance

Pour rappel, la why-provenance d'une donnée correspond aux ensembles des données devant exister conjointement pour témoigner de la présence de cette donnée dans le résultat. La communauté des bases de données s'accorde sur le fait

que la sémantique de la why-provenance est bien renfermée par celle de la how-provenance dans le cas où le semi-anneau pour les annotations est le semi-anneau $(\mathscr{P}(\mathscr{P}(X)), \varnothing, \{\varnothing\}, \cup, \Cup)$ où X est l'ensemble des identifiants des n-uplets et $A \Cup B = \{a \cup b | a \in A \wedge b \in B\}$. En effet, lors d'une union entre deux requêtes q_1 et q_2, les données témoins d'une donnée résultante t ne nécessitent pas d'exister conjointement pour que t apparaisse dans la table résultante. Une union entre les deux ensembles d'ensemble suffit. Lors de la jointure naturelle entre q_1 et q_2, la donnée résultante n'est présente dans le résultat que si nous avons un témoin pour ses projections pour la requête q_1 et la requête q_2.

Je propose ici de vérifier que cette structure est bien un semi-anneau commutatif. Les identifiants des n-uplets sont représentés par un ensemble abstrait muni d'une fonction de comparaison.

```
Hypothesis ids : Type.
Hypothesis Tids : Oeset.Rcd ids.
```

L'ensemble des parties de parties d'identifiants est formalisée à l'aide d'une double application de la fonction `Feset.build` :

```
Notation Why := (Feset.build (FeOA (Feset.build Tids))).
```

L'opération $\Cup$ est définie par la fonction `union_pairwise` (que je ne détaille pas) qui est caractérisée par le lemme suivant :

```
Lemma union_pairwise_mem : ∀ s1 s2 a,
    a ∈ union_pairwise s1 s2 ⟷
      ∃a1 a2, a1 ∈ s1 ⋀ a2 ∈ s2 ⋀ a =SE= Feset.union _a1 a2.
```

Un ensemble `a` est dans l'union `union_pairwise` si et seulement `a` est l'union d'un élément de `s1` et un élément de `s2`.

Avec cette structure, j'obtiens bien le lemme suivant :

```
Lemma Why_is_CSR : CSR Feset.empty (Feset.singleton Feset.empty) Feset.union
        union_pairwise TWhy.
```

ce qui confirme son caractère de semi-anneau commutatif.

Dans le futur, il serait intéressant de montrer que la why-provenance est bien une instance de la how-provenance en Coq. La première étape serait d'implémenter la sémantique de la why-provenance et montrer l'équivalence de celle-ci avec celle de la how-provenance instanciée par le semi-anneau `Why` sur l'algèbre relationnelle positive.

La where-provenance, une autre instance de la how-provenance ?

La where-provenance est le sous-type de provenance qui cherche à connaître de quelle valeur source une valeur finale a été héritée. Pour comprendre les différences entre sa sémantique et celle de la how-provenance, je propose que nous supposions que nous avons un semi-anneau $(A, 0, 1, +, \times)$, qui quand celui-ci instancie la how-provenance, permet de englober la sémantique de where-provenance.

Organe	Gène muté	How-pr
Sein	BRCA1	a

Table 4.36 – NCBIGene (How-provenance)

Organe	How-pr
Sein	a

Table 4.37 – projection sur l'attribut "Organe"

Organe	Gène muté
Sein^x	BRCA1^y

Table 4.38 – NCBIGene (Where-provenance)

Organe
Sein^x

Table 4.39 – projection sur l'attribut "Organe" (Where-provenance)

Dans les **Tables 4.36 et 4.37**, nous proposons d'instancier par un élément de A l'annotation de notre n-uplet et observer sa propagation pour une projection sur l'attribut "Organe". Les **Tables 4.38 et 4.39** correspondent à la même application de requête mais cette fois, nous annotons la table avec la where-provenance : x est l'annotation de l'emplacement pour "Sein" et y pour "BRCA1".

Nous observons que les annotations restent les mêmes lors de la projection pour les **Tables 4.36 et 4.37** alors que pour les **Tables 4.38 et 4.39**, l'annotation se fait au niveau de la case et les annotations sources et finales sont donc différentes. On perd l'information y.

La sémantique de la how-provenance n'est pas assez précise dans ce cas contrairement à la where-provenance qui fait la différence. Il n'est donc pas possible de capturer toutes les subtilités de la sémantique de la where-provenance grâce aux semi-anneaux. Cette différence pose question sur la façon d'implémenter la sémantique de la where-provenance et de chercher à savoir si la sémantique de la how-provenance peut être englobée par celle de la where-provenance.

5 - Where-provenance

Après avoir constaté que la sémantique de la where-provenance ne pouvait être totalement capturée à l'aide des structures de semi-anneaux et de semi-modules, ce chapitre se concentre sur la formalisation de la sémantique de la where-provenance dans le cadre de l'algèbre relationnelle. Je propose aussi la preuve d'une propriété vérifiée par la where-provenance et non par la how-provenance. L'objectif est de mieux comprendre les mécanismes de celle-ci et de les comparer à ceux de la how-provenance. De plus, cette formalisation permettra également d'analyser les limites de la where-provenance par rapport à la how-provenance et de proposer des pistes pour améliorer la capture des informations de provenance dans les requêtes dans le **Chapitre 6**.

5.1 . Where-provenance

La where-provenance a pour objectif de suivre la propagation d'une valeur lors d'une application de requêtes. Pour chaque valeur v d'une donnée obtenue par l'application d'une requête, la where-provenance donne les annotations des cases ayant engendré cette valeur v. Les annotations pour la where-provenance se placent au niveau des cases de la table (et non des lignes).

Pour bien comprendre comment cette propagation s'effectue, nous devons introduire la notion d'emplacement. Un emplacement d'une case c est un triplet constitué de l'attribut associé à c, le n-uplet qui la contient et le nom de la relation où on trouve c. Un emplacement est donc la localisation de la case c. Par exemple, prenons la relation r_0 représentée par la **Table 5.1** - du point de vue des annotations - et la **Table 5.2** - du point de vue des emplacements. La case contenant la valeur 3 est annotée par a_2 et se trouve à l'emplacement (B, t_1, r_0).

	A	**B**	**C**
t_1	1^{a_1}	3^{a_2}	2^{a_3}
t_2	8^{b_1}	13^{b_2}	0^{b_3}

Table 5.1 – relation r_0 : annotations

	A	**B**	**C**
t_1	$1:\{(A, t_1, r_0)\}$	$3:\{(B, t_1, r_0)\}$	$2:\{(C, t_1, r_0)\}$
t_2	$8:\{(A, t_2, r_0)\}$	$13:\{(B, t_2, r_0)\}$	$0:\{(C, t_2, r_0)\}$

Table 5.2 – relation r_0 : emplacements

Lorsqu'on effectue sur cette table une projection sur l'attribut **B** puis qu'on le renomme en **D** (**Table 5.4**), la valeur de cette case est propagée d'abord à l'em-

	B
t_1^{int}	3^{a_2} :$\{(B, t_1^{intr}, r_{int})\}$
t_2^{int}	13^{b_2} :$\{(B, t_2^{int}, r_{int})\}$

Table 5.3 – $r_{int} = \pi_B(r_0)$: annotations et emplacements

	D
t_1^f	3^{a_2} :$\{(D, t_1^f, r_f)\}$
t_2^f	13^{b_2} :$\{(D, t_1^f, r_f)\}$

Table 5.4 – $r_f = \rho_{B:=D}(\pi_B(r_0))$: annotations et emplacements

placement (B, t_1^{int}, r_{int}) puis (D, t_1^f, r_f), la case du n-uplet t_1^f (voir **Table 5.3**). L'annotation a_2 se propage donc dans cette nouvelle case.

On remarque que la propagation des annotations pour la where-provenance est très liée à la notion d'emplacement. Dans les tables sources, une annotation peut être identifiée par l'emplacement de la case qui la contient.

Dans ce chapitre, les annotations pour lesquelles j'ai opté pour ma formalisation de la where-provenance sont dans une structure hybride entre les annotations de la where-provenance et les emplacements.

5.1.1 . Représentation des annotations et des emplacements

Dans la suite, je noterai les annotations de la where-provenance $\text{annot}_{\text{whr}}$ pour éviter toute confusion avec les annotations que je crée pour ma formalisation.

Je me suis donnée deux critères pour les annotations que je construis :

— pouvoir de déduire quelles sont les $\text{annot}_{\text{whr}}$ associées pour toutes les cases d'une table source ou résultante d'une application d'une requête

— être des annotations au niveau des lignes et non sur les cases.

Ce deuxième critère a pour but de pouvoir comparer plus simplement les annotations de ce chapitre avec les annotations de la how-provenance.

Tables sources

Commençons par les tables sources. J'associe une variable unique à chaque couple (n-uplet d'une table source r, relation r). Par exemple, pour la **Table 5.1**, nous associons au n-uplet t_1 et la table r_0 la variable x, et au n-uplet t_2 et la table r_0, la variable y. Un n-uplet ayant hérité une valeur d'un n-uplet t d'une certaine table r sera lié à la variable associée à t et r.

Si nous annotons chaque ligne par la variable associée à celle-ci, ce n'est pas suffisant pour distinguer les $\text{annot}_{\text{whr}}$ de deux cases d'une même ligne. Si je souhaite pouvoir retrouver l'annotation d'une case, j'ai besoin de savoir quel est l'attribut lié à cette case. Pour chaque ligne, l'annotation est donc le couple formé de la variable correspondant à la ligne et la sorte de la relation.

Je note $x : A$, l'annotation qui à la variable x associe à l'ensemble d'attributs A.

La case (B, t_1, r_0) est donc identifiée par l'annotation x : B (dans la **Table 5.5**).

	A	B	C	Annot
t_1	1	3	2	x : A,B,C
t_2	8	13	0	y : A,B,C

Table 5.5 – relation r_0 : annotation version 1

Cette annotation permet de déduire l'emplacement de la case mais aussi l'annot$_{\text{whr}}$ en conservant en mémoire à quel couple (n-uplet, table source) correspond la variable x.

Propagation

Plaçons-nous maintenant après l'application d'une requête de l'algèbre relationnelle positive sans renommage.

Prenons une ligne t d'une table résultante r_2 contenant une case c associée à l'attribut a. Si cette case est héritée d'une case c' de la ligne t' d'une table source r_1, alors les annot$_{\text{whr}}$ de la case c' sont propagées à la case c. De plus, comme on exclut le renommage pour l'instant, l'attribut associé à la case c' est a. Il faut donc que la variable associée au couple (r_1, t') apparaisse dans l'annotation de la ligne t et soit associée à a. Cela est valable pour chaque table source et chaque n-uplet dont la case c hérite. La partie de l'annotation de la ligne t attachée à la case c est donc constituée des variables associées à ces couples (table source,n-uplet) dont c hérite et ces variables sont associée à l'attribut a. Pour obtenir l'annotation complète de la ligne t, il faut réitérer ce raisonnement sur chacune des cases contenues dans t.

On remarque qu'une variable peut être associée à plusieurs attributs. Lorsque nous généralisons aux annotations de données résultantes d'une requête, les annotations sont donc un ensemble de couples de variables et d'ensembles d'attributs. On fera bien attention à ce que chaque variable n'apparaisse qu'une fois au plus dans une annotation. Pour permettre une meilleure compréhension, je note x_1 : A_1|..|x_n : A_n, l'annotation qui à la variable x_1 associe à l'ensemble d'attributs A_1, ..., x_n à l'ensemble A_n. On note ici que chacune des variables est deux à deux disjointes.

Cette propagation des annotations permet de maintenir nos deux critères. Pour une case c associée à l'attribut a d'un n-uplet résultant t, on identifie les relations sources et les lignes des cases dont la case c a hérité grâce aux variables associées à l'attribut a dans l'annotation de la ligne t et ces cases héritées se trouvent dans la colonne a.

Pour illustrer la propagation des annotations, on prend l'exemple (**Table 5.6**) de la projection sur les attributs A et B sur la table source 5.1. La ligne t_3 est annoté x : A,B car les deux cases de t_3 sont héritées des cases contenues dans les colonnes A et B du n-uplet t_1.

Si on cherche à connaître à partir des annotations de t_3 de quelle case la case

	A	B	Annot
t_3	1	3	x : A,B
t_4	8	13	y : A,B

Table 5.6 – $r_1 = \pi_{A,B}(r_0)$: annotations version 1

(A, t_3, r_1) est héritée, on regarde les variables associées à l'attribut A pour le n-uplet t_3 : ici il s'agit de la variable x. La case (A, t_3, r_1) a donc hérité de la case (A, t_1, r_0) et on peut en déduire l'annot$_{\text{whr}}$.

Cas de l'opérateur de renommage

L'ajout de l'opérateur de renommage nécessite de modifier les annotations.

	A	D	Annot
t_5	1	3	x : A,B?,D?
t_6	8	13	y : A,B?,D?

Table 5.7 – $r_2 = \rho_{B:=D}(\pi_{A,B}(r_0))$: annotations version 1

Après le renommage de l'attribut B en D dans la **Table 5.7**, la case (D, t_5, r_2) doit-elle être annotée par l'annotation x : B (on garde l'attribut de la case source) ou x : D (on prend l'attribut de la case résultante) ou encore les deux ? Aucune des ces solutions n'est suffisante : il faut qu'on puisse associer la colonne contenant la case source à la colonne de la case résultante. Pour cela, on modifie nos annotations : à chaque variable est maintenant associée un ensemble de couples d'attributs. Le premier élément du couple correspond à la colonne de la case courante et la seconde à la colonne de la table source d'où provient la case.

Ainsi reprenons les **Tables 5.5 et 5.7**.

	A	B	C	Annot
t_1	1	3	2	x : (A,A),(B,B),(C,C)
t_2	8	13	0	y : (A,A),(B,B),(C,C)

Table 5.8 – relation r_0 : annotation version 2

	1	D	Annot
t_5	1	3	x : (A,A),(D,B)
t_6	8	13	y : (A,A),(D,B)

Table 5.9 – $r_2 = \rho_{B:=D}(\pi_B(r_0))$: annotations version 2

La case (B, t_5, r_2) est donc annotée par x : (D, B) où D désigne la colonne de la table actuelle liée à la case et B l'attribut de la table source. On peut retrouver l'annot$_{\text{whr}}$ de cette case en regardant quelle est l'annotation de la case correspondant à l'attribut B et au couple associé à x : c'est a_2.

Dans cette formalisation, pour retrouver les $\text{annot}_{\text{whr}}$ d'une case c associé à un attribut A et à la ligne t d'une table résultante, on doit :

1. regarder l'annotation de la ligne t
2. récupérer les couples (variable, attribut) (x_1, B_1), .. , (x_n, B_n) tel que les variables x_i sont dans l'annotation de t et sont associées à un ensemble d'attributs contenant un couple de la forme (A, B_i)
3. pour chaque variable x_i, identifiées à quel couple de (n-uplet,relation) celle-ci est associée. Ils sont noté t_i et r_i
4. en déduire que la case c a hérité des cases aux emplacements (B_i, t_i, r_i)
5. associer aux emplacements leur $\text{annot}_{\text{whr}}$

On a ainsi des annotations regroupées par ligne et non case et on peut en déduire l'$\text{annot}_{\text{whr}}$ pour chaque case.

5.1.2 . Intuition sur l'algèbre relationnelle positive

Cette formalisation se place dans l'algèbre relationnelle positive dans le cadre d'une sémantique ensembliste.

Pour illustrer les différentes requêtes possibles, nous proposons les **Tables 5.10, 5.11 et 5.12** représentant des patients, leur diagnostique et leur traitement. On abrège parfois dans la suite "Patient" par l'abréviation "P", "Maladie" par "M", "Sickness" par "S" et "Trait" (traitement) par "T". Dans les annotations, un attribut qui apparaît deux fois dans un même couple d'attributs (par exemple, (A, A)) sera abrégé en A^2.

Requête "Relation"

	P.	Maladie	Annot
t_1	P.1	Rhume	a : P^2,M^2
t_2	P.51	Otite	b : P^2,M^2
t_3	P.59	Angine	c : P^2,M^2
t_4	P.100	Otite	d : P^2,M^2

Table 5.10 – relation r_3

	P.	Maladie	Annot
t_5	P.51	Otite	e : P^2,M^2
t_6	P.67	Angine	f : P^2,M^2

Table 5.11 – relation r_4

	Patient	Trait	Annot
t_7	P.1	Goutte	g : P^2,T^2
t_8	P.59	AB [1]	h : P^2,T^2

Table 5.12 – relation r_5

Chaque n-uplet d'une table source est associé une variable unique. Dans la **Table 5.10**, au n-uplet t_1 est associé la variable a, à t_2, b, à t_3, c, et à t_4, d. Les

couples d'attributs associés à ces variables sont les couples de la forme A_i^2 où les A_i sont les attributs de la sorte de la relation. Aucun autre attribut n'apparaît. Ainsi dans les **Tables 5.10 et 5.11**, les couples d'attributs liés aux variables sont P^2 et M^2, et pour la **Table 5.12**, ils sont P^2 et T^2.

Opérateur n-uplet vide

	Annot
t_9	1 : $\varnothing$

Table 5.13 – $\epsilon_{<>}$

Pour l'opérateur ϵ, le n-uplet vide ne provient d'aucune table. C'est donc un n-uplet qui est spontanément créé au cours de l'application de la requête. Il est donc important d'annoter ce n-uplet d'une façon différente pour conserver cette information. Il est donc annoté par l'annotation particulière $1 : \varnothing$. Le 1 n'est pas une variable et apparaît seulement lorsqu'une donnée n'est pas associée à un n-uplet d'une table source. Il permet d'indiquer que le n-uplet vide n'est hérité d'aucune table et ne dépend d'aucune table source. $\varnothing$ indique que le n-uplet n'a aucun attribut associé.

Renommage

	P.	**Sickness**	**Annot**
t_1	P.1	Rhume	a : P^2,(S,M)
t_2	P.51	Otite	b : P^2,(S,M)
t_3	P.59	Angine	c : P^2,(S,M)
t_4	P.100	Otite	d : P^2,(S,M)

Table 5.14 – renommage $\rho_{Maladie:=Sickness}(r_3)$

Lors d'un renommage, les valeurs héritées restent les mêmes, ce sont seulement les noms des colonnes qui changent. L'opérateur de renommage r joue seulement sur le premier élément des couples d'attributs dans les annotations. Lorsqu'on renomme un attribut a en b, on modifie les annotations en remplaçant le premier élément des couples par b s'il est égal à a. Par exemple, dans la **Table 5.14**, l'attribut "Maladie" est renommé en "Sickness". Les annotations de la table passent de la forme x : P^2, M^2 à x : P^2, (S,M).

Projection

	Maladie	Annot
t_{10}	Rhume	a : M^2
t_{11}	Otite	b : M^2 | d : M^2
t_{12}	Angine	c : M^2

Table 5.15 – projection $\pi_{Type}(r_3)$

L'ensemble des valeurs héritées lors une projection sur un ensemble d'attributs A sont seulement celles contenues dans les colonnes de A de la table de la sous-requête. Ainsi, l'annotation d'un n-uplet t résultant doit correspondre au regroupement des annotations des n-uplets de la sous-requête dont la projection sur A est égale à t et qui sont tronquées pour ne garder que les éléments dont le premier élément du couple d'attributs associé à une variable soit un attribut qui appartient à l'ensemble A. On observe dans la **Table 5.15** que la valeur du n-uplet t_{10} est héritée de la valeur de t_1 contenu dans la deuxième colonne. L'annotation de cette case (a : M^2) est conservée. Cependant, la valeur de l'attribut "Patient" n'est pas héritée, on ne conserve pas a : P^2.
Il est possible pour la projection qu'une case hérite des annotations de plusieurs n-uplets de départ. Par exemple, la valeur du n-uplet t_{11} est elle héritée de t_2 et t_4 de la **Table 5.10** donc l'annotation complète de t_{11} est b : M^2|d : M^2.

Sélection

	Patient	Maladie	Annot
t_3	P.59	Angine	c : P^2,M^2

Table 5.16 – sélection $\sigma_{Type=Angine}(r_3)$

Pour la sélection, seules certaines lignes sont héritées de la table associée à la sous-requête mais les valeurs de ces lignes ne changent pas. On garde donc les mêmes annotations si le n-uplet est conservé. Le n-uplet t_3 est hérité de la **Table 5.10** et vérifie la condition, donc il conserve donc l'annotation c : P^2,M^2. Les autres n-uplets ne vérifient pas la condition donc ils n'apparaissent pas dans la table résultante.

Jointure naturelle

Un n-uplet résultant t issu de la jointure naturelle hérite des cases (contenues dans les sous-requêtes) des projections de t sur les sortes des sous-requêtes.

	Patient	**Maladie**	**Trait**	**Annot**
t_{13}	P.1	Rhume	Goutte	a : P^2,M^2 | g : P^2,T^2
t_{14}	P.59	Angine	AB	c : P^2,M^2 | h : P^2,T^2

Table 5.17 – jointure naturelle($r_3 \bowtie r_5$)

Un n-uplet résultant t issu de la jointure naturelle hérite des cases (contenues dans les sous-requêtes) des projections de t sur les sortes des sous-requêtes. Les annotations conservées pour ce n-uplet sont donc celles des deux n-uplets égaux aux projections de t sur les sortes des sous requêtes. Par exemple, le n-uplet t_{13} de la **Table 5.17** est le résultat de la jointure du n-uplet t_1 de la **Table 5.10** et du n-uplet t_7 de la **Table 5.12**. Les annotations du n-uplet t_{13} regroupent les annotations de t_1 et t_7 donc : l'annotation complète est a : P^2,M^2|g : P^2,T^2.

Union

	Patient	**Maladie**	**Annot**
t_1	P.1	Rhume	a : P^2,M^2
$t_{2,5}$	P.51	Otite	b : P^2,M^2 | e : P^2,M^2
t_3	P.59	Angine	c : P^2,M^2
t_4	P.100	Otite	d : P^2,M^2
t_6	P.67	Angine	f : P^2,M^2

Table 5.18 – union $r_3 \cup r_4$

Chaque n-uplet issu de l'union de deux sous-requêtes est hérité d'un n-uplet contenu dans seulement une des sous-requêtes ou dans chacune des sous-requêtes. L'annotation d'un n-uplet résultant est donc soit l'annotation du n-uplet dont il a hérité dans le premier cas soit le regroupement des annotations des deux n-uplets dans le deuxième.
Par exemple, les valeurs du n-uplet $t_{2,5}$ de la **Table 5.18** sont héritées des n-uplets t_2 et t_5 donc l'annotation de ce n-uplet est le regroupement de b : P^2,M^2 et e : P^2,M^2 donc b : P^2,M^2|e : P^2,M^2. Le n-uplet t_1 de la **Table 5.18** est lui hérité de t_1 de la **Table 5.10** seulement et donc son annotation (a : P^2,M^2) est conservé.

Remarque

Les n-uplets apparaissant dans une table résultante de l'application d'une requête possèdent une annotation non vide. Chaque n-uplet a une annotation dont il a hérité d'une autre table contenant soit une ou plusieurs variables soit un 1. Les n-uplets qui n'apparaissent pas dans une table résultante sont les n-uplets annotés par l'ensemble vide. On a ainsi une similitude entre ces annotations et celle de la how-provenance. En effet, dans chacun des cas, les n-uplets n'étant pas présents dans la table ont une annotation particulière et différente de celle que l'on trouve pour les n-uplets présents dans les tables.

Le regroupement de deux annotations n'est pas exactement l'union classique sur les ensembles. En effet, si on veut regrouper l'annotation c : P^2 et c : M^2, nous voulons avoir c : P^2,M^2 car chaque variable de l'annotation n'apparaît qu'une fois. L'union se fait seulement entre les ensembles de couples d'attributs lorsqu'on regroupe pour une même variable. Ainsi l'annotation résultante d'un regroupement entre deux annotations contient toutes variables des deux annotations et associe à chacune de ces variables x soit l'union de l'ensemble des couples associé à x si x est présents dans les deux annotations soit l'ensemble de couples d'attributs associé à x de l'annotation qui contient x.

5.1.3 . Sémantique de la where-provenance

Toujours dans la volonté de mieux comparer la how-provenance avec la where-provenance, on représente la sémantique sous la forme d'une fonction des n-uplets vers les annotations, que nous nommeront ici aussi `Whr`-relation. Ces fonctions sont similaires aux K-relations du le chapitre précédent : elles sont totales et les n-uplets non présents dans la table sont annotés une annotation particulière, l'ensemble vide.

Pour faciliter la compréhension, nous posons les notations suivantes :

1. l'ensemble $VarOne$: $VarOne = \{variables\} \cup \{1\}$
2. l'ensemble des annotations $\mathbf{Annot}_{\text{whr}}$:

$$\mathbf{Annot}_{\text{whr}} = \mathcal{P}(\{VarOne \times \mathcal{P}(\{attributs\} \times \{attributs\})\})$$

   où $\mathcal{P}$ est l'ensemble des parties d'un ensemble pris en argument
3. l'opération Var : $\mathbf{Annot}_{\text{whr}} \rightarrow VarOne$ définie comme suit :

$$Var(A) = \{x | \exists (x, B) \in A\}$$

   où A est une annotation
4. l'opération π_op : $\{attributs\} \times \mathbf{Annot}_{\text{whr}} \rightarrow \mathbf{Annot}_{\text{whr}}$ définie comme suit :

$$\begin{aligned}\pi\_op(E, F) = &\{(y, G) | \exists (y, H) \in F, G \subseteq H, \\ &\forall (a_1, a_2) \in H, a_1 \in E \Leftrightarrow (a_1, a_2) \in G\}\end{aligned}$$

5. l'opération ρ_op qui à une fonction de renommage et une annotation associe une annotation définie comme suit :

$$\begin{aligned}\rho\_op(r,F) \quad &= \{(y,G)|\exists(y,H)\in F,\\ &\quad G\in\mathcal{P}(\{attributs\}\times\{attributs\})\wedge\\ &\quad \forall(a_1,a_2),(a_1,a_2)\in H\Leftrightarrow(a_1[r],a_2)\in G\}\end{aligned}$$

6. l'opération de regroupement $\uplus$ entre deux annotations quelconques :

$$\begin{aligned}A\uplus B =&\{(y,C\cup D)|\exists(y,C)\in A,\exists(y,D)\in B\}\\ &\cup\{(y,C)|\exists(y,C)\in A,y\notin Var(B)\}\\ &\cup\{(y,C)|\exists(y,C)\in B,y\notin Var(A)\}\end{aligned}$$

7. l'opération $\sqcup$ entre deux ensembles d'annotations quelconques :
   $A\sqcup\varnothing=\varnothing\sqcup A=\varnothing$ où A est un ensemble
   $A\sqcup B=A\cup B$ si A et B sont des ensembles non vides

La sémantique de la where-provenance se décompose comme suit :

Définition 5.1: Sémantique

— $[\![r]\!]_{whr}=t\mapsto$ annot (r, t)
où $annot$ est la fonction qui associe une annotation à chaque n-uplet t d'une relation source r
— $[\![\epsilon_{<>}]\!]_{whr}=t\mapsto\begin{cases}\{(1,\varnothing)\} & \text{si } t=<>\\ \varnothing & \text{sinon}\end{cases}$
— $[\![\rho_r(q)]\!]_{whr}=t\mapsto\rho\_op(r,[\![q]\!]_{whr}(t[r^{-1}]))$
— $[\![\pi_A(q)]\!]_{whr}=t\mapsto\biguplus_{t'|t=t'_{|A}}\pi\_op(A,[\![q]\!]_{whr}(t'))$
— $[\![\sigma_f(q)]\!]_{whr}=t\mapsto\begin{cases}[\![q]\!]_{whr}(t) & \text{si } eval(f,t)=\top\\ \varnothing & \text{sinon}\end{cases}$
où $eval$ evalue la formule logique f sur le n-uplet t
— $[\![q_1\bowtie q_2]\!]_{whr}=t\mapsto[\![q_1]\!]_{whr}(t_{|\mathbf{Attr_q}(q_1)})\sqcup[\![q_2]\!]_{whr}(t_{|\mathbf{Attr_q}(q_2)})$
— $[\![q_1\cup q_2]\!]_{whr}=t\mapsto[\![q_1]\!]_{whr}(t)\uplus[\![q_2]\!]_{whr}(t)$

Dans cette sémantique, on remarque que les variables de l'annotation d'un n-uplet t avant l'application d'une projection sont toutes présentes dans l'annotation de $t_{|A}$ (après la projection sur A). Cela est le cas même si aucun des premiers éléments des couples d'attributs associés à ces variables n'appartient à A. Par exemple, prenons une table ne contenant qu'un n-uplet t ayant pour annotation b : $P^2|e : C^2$. Avec ce qui a été proposé précédemment, lorsqu'on projète sur l'attribut P, on devrait avoir pour $t_{|P}$ l'annotation b : P^2 dans la table résultante. Ici, $t_{|P}$

est annoté par b : P^2|e : $\varnothing$. Pour la variable e, l'annotation e : $\varnothing$ est propagée alors que précédemment ce n'était pas le cas. J'ai fait le choix ici de conserver ces variable dans l'annotation afin d'avoir l'information que le n-uplet lié à e a eu un impact sur le n-uplet résultant.

Nous observons aussi que l'opération effectuée pour la jointure naturelle n'est pas exactement l'opération $\uplus$ entre deux annotations. Elle diffère dans le cas où l'une des deux annotations est l'ensemble vide. Prenons un exemple avec deux sous-requêtes q_1 et q_2 et un n-uplet t. Si la projection de t sur la sorte de q_1 est présente dans la table de la sous-requête q_1 mais celle sur la sorte de q_2 n'est pas présente dans la table de q_2, alors t n'est pas présent dans la table résultant de la jointure de q_1 et q_2. L'annotation doit donc être $\varnothing$. Si on appliquait l'opération $\uplus$ sur les annotations pour ce cas-là, on obtiendrait une annotation non vide.

Comme pour les chapitres précédents, l'opérateur de la projection et l'opérateur du renommage fusionnent dans la formalisation. La sémantique pour cet opérateur est donc :

$$[\![\eta_r(q)]\!]_{whr} = t \mapsto \biguplus_{t'|t=t'[r]} \rho\_op(r, \pi\_op(B, [\![q]\!]_{whr}(t')))$$

où r une fonction de renommage et $B = Ent(r)$ l'ensemble d'entrée de r.

5.2 . Formalisation abstraite de la where-provenance

Dans cette section, je présente ma formalisation abstraite de cette sémantique en Coq. J'oppose dans cette section la formalisation des K-relations et de la sémantique de la how-provenance avec les `Whr`-relations et la sémantique de la where-provenance.

5.2.1 . Annotations

Les annotations que nous avons présentées précédemment, s'écrivent sous la forme : $x_1 : (A_1^1, B_1^1), .., (A_{m_1}^1, B_{m_1}^1)|..|x_n : (A_1^n, B_1^n), .., (A_{m_n}^n, B_{m_n}^n)$ où les x_1, .., x_n sont deux à deux distincts. On peut donc formaliser les annotations de ce chapitre à l'aide de fonctions partielles qui à une variable $VarOne$ associe un ensemble de couples d'attributs. Parmi les formalisations des fonctions partielles proposées dans la **Sous-section 3.4.1**, le module `Femap3` est le plus approprié pour englober les différentes opérations appliquées sur les annotations de whr.

Pour une annotation `a`, l'ensemble des variables de l'annotation est donc donné par la fonction `base a` et l'ensemble des couples d'attributs d'une variable `x` correspond au coefficient de `x` pour l'annotation `a`.

L'élément `zero` étant l'élément renvoyé par défaut par la fonction `coeff` lorsque les variables n'apparaissant pas dans la `base`, il prend ici la valeur de l'ensemble vide. Pour ce qui est des opérations qu'on doit pouvoir appliquer sur ces fonctions partielles, nous avons besoin d'une opération entre deux fonctions partielles qui calcule l'opération $\uplus$ entre deux annotations : elle doit avoir comme

base (donc comme ensemble de variable) l'union des bases des annotations initiales et faire l'union des coefficients. Cela est permis grâce à l'opération `union` de `Femap3`. Pour les opérations ρ_op et π_op, seuls les ensembles de couples d'attributs sont modifiés : nous avons besoin de la fonction `transpo` pour faire des opérations seulement sur les coefficients.

L'ensemble $VarOne$ est formalisé par le type `K` et l'hypothèse `X` correspond à l'ensemble des variables.

```
Hypothesis X : Type.
Inductive K: Type :=
  |One : K
  |Var : X → K.
```

L'ensemble des parties de l'ensemble des couples d'attributs est le suivant :

```
Definition FAA :=
  Fset.build (Oset.mk_pair (OAtt T) (OAtt T)).
```

La fonction `Oset.mk_pair` crée une relation d'ordre total du produit cartésien de deux ensembles à partir de leurs relations d'ordre. `Fset.build` prend une relation d'ordre d'un ensemble et crée l'ensemble des parties de cet ensemble.

Pour formaliser les opérations sur l'ensemble $\mathbf{Annot}_{\mathsf{whr}}$, on instancie le module `Femap3` avec pour ensemble de départ le type `K` et pour ensemble d'arrivée le type `FAA`.

```
Definition PolK := Femap3.build OK OFAA Fset.empty Fset.union Fset_concat.
```

La fonction `Femap3.build` construit l'ensemble des fonctions partielles à partir des relations d'ordres des ensembles d'entrée (`OK`) et de sortie (`OFAA`), de l'élément `zero` (l'ensemble vide `Fset.empty`) et des opérations `plus` (`Fset.union`) et `mul` (`Fset_concat`).

Attardons-nous sur `Fset_concat`. Elle vérifie le lemme :

```
Lemma Fset_mem_concat : ∀ a s1 s2,
    a ∈? Fset_concat s1 s2 = true ⟷
    ∃b,
      (b, snd a) ∈ s1 ⋀ (fst a, b) ∈ s2.
```

`s2` correspond à la fonction de renommage que l'on souhaite appliquer et `s1` un ensemble de couples d'attributs associé à une variable x d'une annotation. La fonction de renommage est ici représentée sous la forme de couples d'attributs : le premier élément est qu'on souhaite renommer et le deuxième est celui qu'on veut à la place. Après application de la fonction de renommage, le deuxième élément (ici `snd a`) d'un couple d'attributs résultant associé à la variable x doit rester le même qu'avant l'application de `s2`. Le premier élément (`b`) avant l'application est modifié en appliquant la fonction de renommage de `s2`, ce qui correspond à chercher dans `s2` quel élément est associé à `b` et l'utiliser comme deuxième élément (`fst a`).

5.2.2 . Whr-relations

L'enregistrement des `Whr`-relations diffère de celui des K-relations de la how-provenance seulement pour le type de sortie de la fonction `f` qui prend le nouveau type des annotations de la where-provenance `femap PolK`.

```
Record Whrel : Type :=
  mk_kr
   {
     f : tupleT → femap PolK;
     support : setT;
     sort_whrel : Fset.set (A T);
   }.
```

En ce qui concerne les invariants que les `Whrel` doivent vérifier pour être bien formées, la seule modification par rapport `Krel_inv` se porte sur l'invariant `finite_supp`. La vérification que l'annotation n'est pas égale à `zero` (`eq OK (f kr t) zeroK`) est remplacée par `is_zero (f kr t) = true`. La fonction `is_zero : femap PolK -> bool` vérifie si la base d'une annotation est équivalente ensemble vide. Son rôle est donc de déterminer si un n-uplet apparaît dans une table. On la distingue de la relation d'ordre des fonctions partielles. Pour qu'il y est équivalence entre une annotation et l'ensemble vide, la relation d'ordre vérifie seulement si les coefficients sont égaux à l'ensemble vide et ne pose aucune condition sur la base. On note que cette relation d'ordre entre fonctions partielles sera utilisée dans la suite lors de la formalisation d'une propriété (voir **Section 5.3**).

J'ai ainsi modifié l'invariant `finite_supp` des K-relations par :

```
finite_supp : ∀ t, Feset.mem t (support kr) = false → is_zero (f kr t) = true;
```

5.2.3 . Sémantique des requêtes

La sémantique est comme pour celle de la how-provenance formalisée par une fonction récursive nommée dans cette partie `eval_prov_whr`. Je définis la formalisation de la sémantique de la where-provenance en comparaison avec les fonctions proposées dans la how-provenance.

La première différence avec la sémantique de la how-provenance est le remplacement de l'élément zero par `empty PolK` qui correspond à l'annotation ensemble vide. Ce remplacement est justifié par le fait que ces deux éléments sont dans les deux cas utilisés pour identifier les éléments non présents dans une table.

Pour l'opérateur du n-uplet vide, le un est remplacé par $1 : \varnothing$ qui est formalisé par `singleton (One X) (emptysetS)`.

Dans la sémantique de l'opérateur union, l'addition entre les deux annotations d'un n-uplet pour les sous-requêtes est remplacée par l'opération `union PolK` entre les deux annotations.

Relations sources

En ce qui concerne les relations sources, nous posons les hypothèses équivalentes à celle proposées dans how-provenance à l'exception de l'hypothèse `instance_prov_equiv` qui suppose que la fonction des annotations des K-relations sources est compatible avec la relation d'équivalence des n-uplets. Si nous faisons cela ici, avec la définition que nous avons de l'égalité sur les fonctions partielles, la compatibilité ne s'applique seulement sur les coefficients. On divise cette supposition en deux : une pour la compatibilité des coefficients et une pour la compatibilité des bases.

```
Hypothesis instance_prov_equiv1 : ∀ r x1 x2 x,
    eq x1 x2 →
    coeff x (f (instance_prov r) x1) ≡ coeff x (f (instance_prov r) x2).

Hypothesis instance_prov_equiv2 : ∀ r x1 x2,
    eq x1 x2 →
    base (f (instance_prov r) x1) =SE= base (f (instance_prov r) x2).
```

Jointure naturelle

La K-relation liée à la jointure naturelle est modifiée de la façon suivante :

```
Definition f_nj_Whrel wr1 wr2:=
  fun t ⇒
    if (labels t ≡? (sort_whrel wr1 ∪ sort_whrel wr2)) &&
        negb (is_zero (f wr1 (mk_tuple (sort_whrel wr1) (dot t)))) &&
        negb (is_zero (f wr2 (mk_tuple (sort_whrel wr2) (dot t))))
    then union
          (f wr1 (mk_tuple (sort_whrel wr1) (dot t)))
          (f wr2 (mk_tuple (sort_whrel wr2) (dot t)))
    else empty.
```

On observe une première différence avec la how-provenance dans la condition du `if then else`. Une condition supplémentaire est ajoutée : les annotations des projections du n-uplet t dans les sous-requêtes ne sont pas équivalentes l'annotation ensemble vide. Dans le cas où cette condition n'est pas satisfaite, la sémantique spécifie que l'annotation résultante de t est l'ensemble vide donc `empty PolK`. Dans la sémantique de la how-provenance, cette vérification n'a pas besoin d'être explicitement faite : le zéro du semi-anneau est absorbant pour la multiplication et donc si l'une des deux annotations est équivalente à zéro, l'annotation résultante est équivalente à zero.

La deuxième différence porte sur l'opération entre les deux annotations dans le cas où les conditions sont satisfaites. Les opérations effectuées entre les annotations de la how-provenance pour les opérateurs union et jointure naturelle sont distinctes, contrairement à la where-provenance où les opérations sont toutes les deux effectuées avec l'opération `union PolK`.

Projection

On remarque ici que les fonctions ρ_op et π_op commutent avec l'opération $\uplus$: nous pouvons donc appliquer d'abord $\uplus$ sur les annotations puis appliquer ρ_op et π_op.

La partie de l'application $\uplus$ est similaire à la fonction des annotations pour la projection présentée dans la how-provenance à quelques modifications près. En effet, pour un n-uplet résultant t d'une requête $\eta_r(q)$, calculer le regroupement de toutes les annotations des n-uplets associés à q donnant t après l'application de la fonction de renommage est comparable à faire l'addition des annotations de ce même groupe de n-uplets dans la how-provenance.

La première modification à apporter est donc le remplacement de l'addition par l'opération `union`. La deuxième différence vient du calcul de la projection-renommage des n-uplets par une fonction de renommage : dans la how-provenance, nous nous étions placés dans l'algèbre relationnelle positive étendue qui suppose que la sorte des n-uplets résultants sont donnés par l'ensemble de sortie de la fonction de renommage, ce qui n'est pas le cas dans l'algèbre relationnelle. Ainsi la fonction de projection est modifiée :

```
Definition projection_prov env las s :=
    mk_tuple T
      (sort_pi s las)
      (fun a ⇒
       match Oset.find a las with
       | Some e ⇒ interp_aggterm_prov env e
       | None ⇒ dot (default_tuple (∅)) a
       end).
```

Si `mk_tuple` prend un deuxième argument similaire à ce que propose la how-provenance, le premier élément diffère : `sort_pi` renvoie l'application de la fonction de renommage `las` sur l'ensemble des attributs de `s`. En effet ici, les attributs de l'ensemble des sorties de la fonction de renommage ne sont pas forcément des attributs qui apparaissent dans la table résultante. On notera ici que la sorte des n-uplets pour cette Whr-relation est `sort_pi`.

L'application des fonctions ρ_op et π_op est effectuée en appliquant l'opération supplémentaire `transpo` (en orange) à la partie de $\uplus$ (en violet) :

```
Definition f_pi wr (env:env_prov) s := fun t' ⇒
  transpo (renaming_coeff s)
  (fold_left (fun (x : femap PolK) (y : A) ⇒ union (f wr y) x)
     (filter
        (fun t ⇒ Oeset.eq_bool t' (projection_prov (env_pt env t) s (sort_whrel wr)))
        (Feset.elements (support wr))) empty)).
```

La fonction `renaming_coeff` convertit la fonction de renommage de type `list (select T)` en un ensemble de couples d'attributs pour être compatible avec la fonction `Fset_concat`.

Les supports et les sortes des Whr-relations (exceptée pour la sorte de la projection), les environnements et leur fonctions d'équivalence sont définis de manière similaire à la how-provenance.

5.2.4 . Bonne formation des requêtes

La fonction de bonne formation `well_formed` des requêtes diffère de celle de la how-provenance pour le cas de la projection-renommage et le cas des fonctions d'agrégation.

Comme nous nous plaçons dans l'algèbre relationnelle positive, le cas des fonctions d'agrégation disparaît et donc l'argument booléen est supprimé. La bonne formation pour la projection-renommage est définie différemment dans l'algèbre relationnelle positive :

```
| Q_Pi s q ⇒
  well_formed q && forallb filter_pi s &&
  rename_sort s q && rename_all_diff s
```

La requête n'est pas bien formée si la sous-requête q est mal formée. `filter_pi` impose que la fonction de renommage prenne en entrée seulement des attributs et pas des expressions d'attributs. `rename_sort` vérifie que les attributs en entrée de la fonction de renommage sont bien inclus dans la sorte de la sous-requête q. Enfin la dernière condition `rename_all_diff` contraint les attributs en entrée de la fonction de renommage d'être deux à deux distincts et il en va de même pour les attributs en sortie. Dans la formalisation de la how-provenance, on ne vérifiait que les attributs était deux à deux distincts dans l'ensemble d'arrivée.

5.2.5 . Preuves générales

Pour les mêmes raisons que pour la how-provenance, j'ai prouvé des théorèmes généraux similaires à ceux de la **Sous-section 4.5.6**. Cependant pour la compatibilité avec l'équivalence des n-uplets et celle des environnements pour la fonction des annotations, je vérifie que les coefficients des fonctions d'annotations sont équivalents mais aussi que les bases sont équivalentes.

L'équivalence **(Équivalence sorte-requête-Whr-relation)** est prouvé par induction sur la taille de la requête :

```
Theorem sort_whrel_basesort : ∀ env q,
    sort q ≡ sort_whrel (eval_prov_whr env q).
```

Nous avons comme corollaire que `sort_whrel` est compatible avec l'équivalence des environnements.

Dans le cadre de la where-provenance, tous les autres théorèmes ont nécessité d'être prouvé simultanément (à cause des interdépendances entre les différentes fonctions) et par induction sur la taille de la requête.

Pour ce qui est des hypothèses supplémentaires de ces théorèmes, on ne fait aucune supposition sur la bonne formation des requêtes contrairement à la how-provenance. Nous posons cependant pour les théorèmes **(Non appartenance - annotation nulle)** et **(Équivalence sorte-n-uplet-Whr-relation)** une hypothèse

supplémentaire sur les environnements. Les Whr-relations présentes dans les environnements doivent avoir leurs fonctions d'annotations compatibles avec l'équivalence des n-uplets, c'est-à-dire que les coefficients des fonctions d'annotations soient équivalents pour des n-uplet équivalents.

```
Definition well_formed_env_slicep (e : group_by * Whr) :=
    let (g, kr) := e in
    ∀ t1 t2,
        Oeset.eq_bool t1 t2 = true →
        equal (f kr t1) (f kr t2) = true.

Definition well_formed_envp (e : env_prov) := Forall well_formed_env_slicep e.
```

On a donc les théorèmes suivants :

```
Theorem Whrel_inv_eval : ∀ env q,
    well_formed_envp env →
    Whrel_inv (eval_prov_whr env q).

Theorem support_eq : ∀ q e1 e2,
    equiv_envp e1 e2 →
    support (eval_prov_whr e1 q) =SE= support (eval_prov_whr e2 q).

Theorem base_env_eq : ∀ e1 e2 q t1 t2,
    equiv_envp e1 e2 →
    eq t1 t2 →
    base (f (eval_prov_whr e1 q) t1) =SE= base (f (eval_prov_whr e2 q) t2).

Theorem coeff_in_eq : ∀ e1 e2 q t1 t2 x a,
    equiv_envp e1 e2 →
    eq t1 t2 →
    a ∈ coeff (Var x) (f (eval_prov_whr e1 q) t1) →
    a ∈ coeff (Var x) (f (eval_prov_whr e2 q) t2).

Theorem eval_formula_prov_eq : ∀ s e1 e2,
    equiv_envp e1 e2 →
    eval_formula_prov eval_prov_whr e1 s = eval_formula_prov eval_prov_whr e2 s.
```

5.2.6 . Formalisation des tables sources

Nous avons précédemment insisté sur l'initialisation des relations sources. Penchons nous sur celle-ci.

Chaque n-uplet t d'une table source est associé à une variable unique x et si la sorte de t est $A_1, .., A_n$, t est annoté par x : A^2_1,..,A^2_n.

Pour formaliser les propriétés sur les tables sources, j'ai tout d'abord défini la fonction `var_to_tuple : X -> option tuple` qui permet d'associer une variable avec le n-uplet avec lequel elle est liée dans les tables sources. Lorsque la fonction renvoie `None` pour une variable, celle-ci n'est associé à aucun n-uplet des tables sources.

On doit donc s'assurer si un n-uplet `t` d'une table source `r` est associé à une variable `p`, l'ensemble des couples d'attributs est comme proposé précédemment :

```
Hypothesis inst_annot1 : ∀ r t p,
    Var p ∈ base (f (instance_prov r) t) →
    coeff (Var p) (f (instance_prov r) t) ≡ double_labels (basesort r).
```

où `double_labels` répète les attributs de `basesort r` pour passer d'un ensemble d'attribut $\{A_1, .., A_n\}$ à l'ensemble $\{(A_1, A_1), .., (A_n, A_n)\}$. On rappelle que `basesort` donne la sorte d'une relation source.

Il faut aussi s'assurer que les n-uplets sources ne sont pas annoté par 1 :

```
Hypothesis inst_annot2 : ∀ t r,
    One ∈? base _(f (instance_prov r) t) = false.
```

J'assure par les deux hypothèses qui suivent que chaque variable apparaît au plus une fois dans une table source et aucune fois si elle n'est pas liée à un n-uplet par `var_to_tuple`.

`inst_annot3` garantit que chaque variable associée à un n-uplet t par `var_to_tuple` n'apparaît pas dans l'annotation d'un n-uplet non équivalent à t.

```
Hypothesis inst_annot3 : ∀ p t t' r,
    var_to_tuple p = Some t →
    Oeset.eq_bool t t' = false →
    Var p ∈? base (f (instance_prov r) t') = false.
```

`inst_annot4` garantit que les variables pour lesquelles `var_to_tuple` renvoie `None` soient bien non associées à un n-uplet t dans les tables sources et donc n'apparaissent dans l'annotation d'un n-uplet d'une table source.

```
Hypothesis inst_annot4 : ∀ p r t,
    var_to_tuple p = None →
    Var p ∈? base (f (instance_prov r) t) = false.
```

On remarque à ce stade que je n'impose pas que pour deux tables différentes r et r', les annotations liées à n-uplet t doivent être différentes. Cette condition n'étant pas nécessaire pour les preuves de la **Section 5.3**, elle est imposée seulement dans le cas concret. Je n'impose pas non plus qu'une variable qui est lié à un n-uplet par `var_to_tuple` apparaisse dans une table source.

5.3 . Propriété sur la where-provenance

Dans cette sous-section, nous nous intéressons à une propriété de la where-provenance qui n'est pas satisfaite par la how-provenance et de la preuve de celle-ci avec ma formalisation de la where-provenance. La propriété est la suivante :

Propriété 5.1: Propriété équivalence-annotation-n-uplets

Existe-t-il une provenance non triviale et une initialisation des annotations des tables sources telles que si deux n-uplets de tables résultantes de l'application de deux requêtes différentes ont des annotations équivalentes, alors ces n-uplets sont équivalents ?

La propriété n'est pas satisfaite par la how-provenance. En effet, un n-uplet $t = (Angine, BCRA1)$ (où $Angine$ est associé à l'attribut $Maladie$) et la projection de t sur l'attribut $Maladie$ ne sont pas équivalents mais leur annotations le sont quelque soit le semi-anneau choisi.

Pour la where-provenance, deux valeurs ayant hérité du même emplacement sont équivalentes car la valeur héritée reste la même que celle de départ. La propriété semble être vérifiée pour la where-provenance.

Nous voyons dans les sous-sections suivantes, comment formaliser cette propriété en Coq et la structure de la preuve de cette propriété.

5.3.1 . Formalisation de la propriété

La propriété à prouver est formalisée comme suit :

```
Theorem equal_tuple_final : ∀ q1 q2 t1 t2,
        well_formed q1 = true →
        well_formed q2 = true →
        is_zero (f (eval_prov_whr nil q1) t1) = false →
        is_zero (f (eval_prov_whr nil q2) t2) = false →
        equal (f (eval_prov_whr nil q1) t1) (f (eval_prov_whr nil q2) t2) = true →
        eq (OTuple T) t1 t2.
```

L'équivalence `equal` entre deux annotations a_1 et a_2 renvoie vraie si et seulement si pour chaque variable (présente ou non dans la `base`), les coefficients associés à cette variable pour a_1 et a_2 sont les mêmes. On remarque ici que nous n'avons pas besoin d'avoir des informations sur la base des annotations pour obtenir la propriété. Comme la propriété concerne seulement les annotations dont les n-uplets sont présents dans les tables résultantes, il faut ajouter que les annotations ne sont pas l'ensemble vide. Les deux autres hypothèses qui sont ajoutées assurent que les requêtes sont bien formées.

5.3.2 . Structure de la preuve de la propriété

Pour prouver que deux n-uplets sont équivalents, on utilise propriété `tuple_eq` du module `Tuple` (**Sous-section 3.3.1**) qui découpe la preuve en deux parties. Dans notre cas, il faut donc :

- prouver que les sortes des n-uplets sont équivalentes sous les mêmes hypothèses
- et prouver que si un attribut a appartient à la sorte de t_1 alors les valeurs associées à a de `t1` et `t2` sont égales

Cela engendre les deux lemmes suivants :

```
Lemma equal_sort_eq : ∀ t1 t2 q1 q2,
  well_formed q1 = true →
  well_formed q2 = true →
  is_zero (f (eval_prov_whr nil q1) t1) = false →
  is_zero (f (eval_prov_whr nil q2) t2) = false →
  equal (f (eval_prov_whr nil q1) t1)
        (f (eval_prov_whr nil q2) t2) = true →
  sort_whrel (eval_prov_whr nil q1) ≡ sort_whrel (eval_prov_whr nil q2).
```

```
Lemma equal_dot_eq : ∀ q1 q2 t1 t2 a,
      well_formed q1 = true →
      well_formed q2 = true →
      is_zero (f (eval_prov_whr nil q1) t1) = false →
      is_zero (f (eval_prov_whr nil q2) t2) = false →
      equal (f (eval_prov_whr nil q1) t1)
            (f (eval_prov_whr nil q2) t2) = true →
      a ∈ sort_whrel (eval_prov_whr nil q1) →
      dot T t1 a = dot T t2 a.
```

Je détaille le déroulement des preuves de ces deux lemmes à l'aide des lemmes définis ci-après. Je présente avant cela les dépendances entre les lemmes pour faciliter la compréhension.

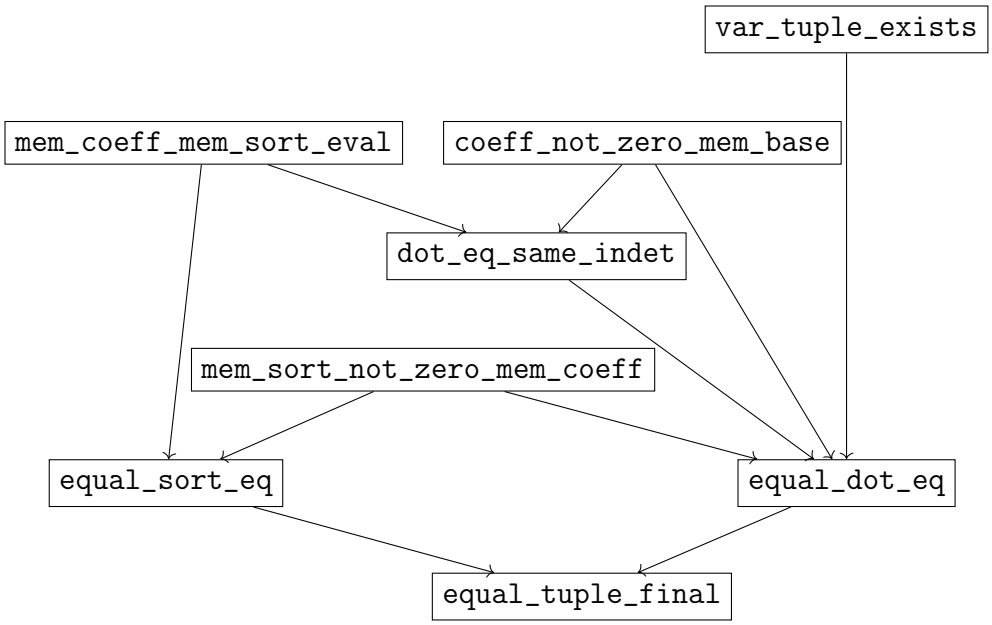


Figure 5.1 – Dépendance des lemmes

Un lemme A dépend d'un lemme B si on a une flèche $B \rightarrow A$ dans le schéma.

Lemme equal_sort_eq

Commençons par le lemme `equal_sort_eq` : deux lemmes auxiliaires `mem_sort_not_zero_mem_coeff` et `mem_coeff_mem_sort_eval` ont été nécessaires pour l'obtenir.

Nous souhaitons montrer que la sorte d'une requête q est équivalente à l'union des premiers éléments des couples d'attributs présents dans les coefficients de l'annotation d'un n-uplet t de la table résultante de q.
En effet, si a est un attribut contenu dans la sorte de la requête q, t a une valeur

v pour cet attribut. Cette valeur est héritée d'une case associée à un attribut b d'un n-uplet d'une table source. Il y a donc dans l'annotation de t une variable qui est associée à un couple d'attributs (a, b). Et inversement, si a apparaît comme premier élément d'un couple d'attributs d'un coefficient de l'annotation de t alors par construction des Whr-relations, a est dans la sorte de q.

La première étape est donc de démontrer que si on a un n-uplet t dans la table résultante d'une requête q, chaque attribut de la sorte de q apparaît dans l'annotation de t en tant que premier élément d'un couple d'attributs dans l'annotation de t associée à une variable .

```
Lemma mem_sort_not_zero_mem_coeff : ∀ e q a t,
    well_formed q = true →
    well_formed_envp e →
    a ∈ sort_whrel (eval_prov_whr e q) →
    is_zero (f (eval_prov_whr e q) t) = false →
    ∃x,
      a ∈ Fset.map fst (coeff (Var x) (f (eval_prov_whr e q) t)).
```

Le lemme nécessite de supposer que la requête et l'environnement sont bien formés.

La deuxième étape est de montrer que si un attribut a est dans un des ensembles de couples d'attributs d'une annotation en tant que premier élément d'un couple alors cet attribut a est dans la sorte de la requête.

```
Lemma mem_coeff_mem_sort_eval : ∀ env q t x a,
    well_formed_weak q = true →
    a ∈ Fset.map fst (coeff (Var x) (f (eval_prov_whr env q) t)) →
    a ∈ sort_whrel (eval_prov_whr env q).
```

On obtient donc que la sorte d'une requête et l'ensemble des premiers éléments des ensembles d'attributs d'une annotation d'un n-uplet de cette requête sont équivalents si l'annotation n'est pas l'ensemble vide.

Il ne reste plus qu'à utiliser l'équivalence entre les coefficients des annotations des deux requêtes, on obtient bien `equal_sort_eq`.

Lemme equal_dot_eq

Pour prouver le second lemme, nous avons besoin de trois lemmes supplémentaires :

—
```
Lemma coeff_not_zero_mem_base : ∀ a x p,
    a ∈ coeff x p →
    x ∈ base p.
```

Ce lemme prouve que si on a un ensemble de couples d'attributs non vide dans le coefficient associé à une variable `x` pour une annotation `p` alors cet attribut est forcément dans la base de `p`.

—
```
Lemma var_tuple_exists: ∀ p env q t,
      Var p ∈ base (f (eval_prov_whr env q) t) →
      ∃t', var_to_tuple p = Some t'.
```

Si une variable p apparaît dans une annotation de la table résultante d'une requête q, il existe un n-uplet t' associé à p par la fonction `var_to_tuple`.

—
```
Lemma dot_eq_same_indet : ∀ env q t p t' a b,
    well_formed q = true →
    well_formed_envp env →
    (a,b) ∈ coeff (Var p) (f (eval_prov_whr env q) t) →
    var_to_tuple p = Some t'→
    dot t' b = dot t a.
```
Pour toute variable p d'une annotation ann liée à un n-uplet t et qui identifie le n-uplet source t', si les coefficients de ann sont non vides pour la variable p alors les valeurs de t et t' coïncident sur les attributs liés à la variable p.

Avec tous ces théorèmes, on peut ainsi déduire à partir de l'hypothèse `a inS sort_whrel (eval_prov_whr nil q1)` que :

— il y a une variable p tel que
`a inS Fset.map fst (coeff (Var p1)`
`(f (eval_prov_whr nil q1) t1))`
(donc il existe un b tel que :
`(a,b) inS coeff (Var p) (f (eval_prov_whr nil q1) t1))`

— comme les coefficients des annotations sont équivalents, on a
`(a,b) inS coeff (Var p) (f (eval_prov_whr nil q2) t2)`

— p est dans la base de `(f (eval_prov_whr nil qi) ti))` où i est soit 1 soit 2

— p est associé à un n-uplet `ti'` d'une table source

— la valeur de `ti` sur l'attribut `a` est égale à celle de `ti'` sur l'attribut `bi` : `dot ti' b = dot ti a`

De plus, comme `var_to_tuple` est une fonction, les `ti'` sont égaux et on obtient ainsi le théorème.

La where-provenance vérifie donc la propriété **(Equivalence-annotations-n-uplets)**.

5.4 . Instanciation

Deux niveaux d'instanciation sont proposées pour la where-provenance. Le premier niveau instancie les structures de données (c'est-à-dire les types et fonctions du module `Tuple`) mais laisse abstrait l'ensemble des variables. Le second niveau propose d'instancier l'ensemble des variables par l'ensemble des couples d'entiers naturels. On cherche dans chacun de ces niveaux d'instanciations à savoir sous quelles hypothèses la propriété **(Equivalence-annotations-n-uplets)** est vérifiée.

5.4.1 . Instanciation 1 : structure de données concrète

On instancie la structure de données de la même façon que proposé dans `Datacert` à l'exception de la fonction d'interprétation d'agrégation qui est elle

instanciée par une fonction renvoyant une valeur par défaut car nous sommes dans l'algèbre relationnelle. Pour les annotations, l'ensemble des variables `X` est abstrait mais les attributs sont eux concrets. La fonction `var_to_tuple` reste donc abstraite.

Je vais m'attarder sur les éléments qui diffèrent avec l'instanciation de la how-provenance. Contrairement à la how-provenance pour laquelle les noms de relations sont des chaînes de caractères, les noms de relations sont représentés par des identifiants dans l'ensemble des entiers naturels.

```
Inductive relname_ : Set := Rel : N → relname_.
```

Il est toujours possible d'associer une chaîne de caractère à chaque identifiant.

Je propose aussi d'avoir des bases de données avec des tables déjà annotées par la where-provenance :

```
Record db_state : Type := mk_state_
  { relnames_ : list relname_;
    instance_prov_ : relname_ → Whrel}.
```

`relnames_` correspond à l'ensemble des noms de relations présents dans la base et `instance_prov_` donne la Whr-relation associée à un nom de table.

Propriété (Equivalence-annotations-n-uplets)

Pour que la propriété **(Equivalence-annotations-n-uplets)** soit vraie dans le cadre de cette instanciation, il faut que les conditions suivantes soient vérifiées :

- la compatibilité de la base et les coefficients de la fonction des annotations des relations sources avec l'équivalence des n-uplets
- les conditions `inst_annot1`, `inst_annot2`, `inst_annot3` et `inst_annot4` sur les tables sources
- les invariants des Whr-relations

Pour que la plupart de ces conditions soient vérifiées, j'ai imposé deux invariants à la fonction `var_to_tuple` et trois fonctions de bonne formation sur les bases de données.

Les invariants de la fonction `var_to_tuple_inv` sont :

```
Record var_to_tuple_inv (db : db_state) : Type := mk_f_ind
  { inst_monome3 : ∀ x t t' r,
      var_to_tuple db x = Some t → Oeset.eq_bool t t' = false →
      Var x ∈? base (f (instance_prov_ db r) t') = false;
    inst_monome4 : ∀ x r t,
      var_to_tuple db x = None →
      Var x ∈? base (f (instance_prov_ db r) t) = false;
    }.
```

Cela remplit déjà les 2 conditions `inst_annot3` et `inst_annot4`.

Les fonctions de bonne formation des bases de données sont :

- `well_formed_db` qui assure que la sorte des n-uplets du support d'une relation r est bien équivalente à la sorte de r.

```
Definition well_formed_db db r :=
  Feset.for_all (fun x ⇒ Fset.equal (labels x) (sort_whrel (instance_prov_ db r))
                (support (instance_prov_ db r)).
```

— `well_formed_db1` vérifie que pour chaque n-uplet d'une table source, la base associée ne peut contenir que des variables et pas de un et que l'ensemble des couples d'attributs associés à chaque variable est bien l'ensemble des couples dont les deux éléments sont identiques et sont dans la sorte de la table

```
Definition well_formed_db1 (db : db_state) r t :=
  Feset.for_all
        (fun x ⇒
         match x with
         |One ⇒ false
         |Var p ⇒
            Fset.equal (coeff (Var p) (f (instance_prov_ db r) t))
                       (double_labels (sort_whrel (instance_prov_ db r)))
          end) (base (f (instance_prov_ db r) t)).
```

— `well_formed_db2` vérifie qu'un n-uplet n'étant pas dans le support d'une Whr-relation source a bien l'annotation ensemble vide.

```
Definition well_formed_db2 (db : db_state) r t :=
  (t ∈? support (instance_prov_ db r)) ||
  Feset.is_empty (base (f (instance_prov_ db r) t)).
```

Si les conditions `well_formed_db` et `well_formed_db2` sont vérifiées, les invariants sur les Whr-relations sont vérifiées :

```
Lemma instance_prov_inv : ∀ db r,
    well_formed_db db r = true →
    (∀ t, well_formed_db2 db r t = true) →
    Whrel_inv (instance_prov_ db r).
```

Si la condition `well_formed_db1` est vérifiée, nous avons que les 2 conditions `inst_annot1` et `inst_annot2` sont vérifiées.

La compatibilité avec l'équivalence des n-uplets pour la base des fonctions d'annotations des tables sources doit être posée en hypothèse car l'ensemble des variables est abstrait. Avec cette hypothèse et la condition `well_formed_db1`, nous avons la compatibilité pour les coefficients avec l'équivalence des n-uplets.

```
Lemma instance_prov_coeff_eq_mem : ∀ db r,
    (∀ t, well_formed_db1 db r t = true) →
    ∀ t1 t2 x a,
    eq t1 t2 →
    base (f (instance_prov_ db r) t1) =SE=
    base (f (instance_prov_ db r) t2) →
    a ∈ coeff x (f (instance_prov_ db r) t1)→
    a ∈ coeff x (f (instance_prov_ db r) t2).
```

Nous obtenons donc la propriété **(Equivalence-annotations-n-uplets)** sous ces hypothèses :

```
Theorem equal_tuple_final : ∀ db q1 q2 t1 t2,
    (∀ r, well_formed_db db r = true) →
    (∀ r t, well_formed_db1 db r t = true) →
    (∀ r t, well_formed_db2 db r t = true) →
    var_to_tuple_inv db →
    (∀ r t1 t2,
    eq t1 t2 →
    base (f (instance_prov_ db r) t1) =SE=
    base (f (instance_prov_ db r) t2)) →
    well_formed q1 = true →
    well_formed q2 = true →
    is_zero (f (eval_prov db nil q1) t1) = false →
    is_zero (f (eval_prov db nil q2) t2) = false →
    Femap3.equal (f (eval_prov db nil q1) t1) (f (eval_prov db nil q2) t2) = true →
    eq t1 t2.
```

5.4.2 . Instanciation 2 : ensemble des variables

Dans ce deuxième niveau d'instanciation, l'ensemble des variables est représenté par l'ensemble des couples d'entiers naturels.

On part d'une base de données sans annotations qu'on annote par la where-provenance dans un deuxième temps :

```
Record db_state_ : Type := mk_state
  { _relnames : list relname_;
    _basesort : N → Fset.set (A TNullExpr);
    _instance : N → Feset.set (Feset.build (OTuple TNullExpr))}.
```

`_relnames` donne les différentes tables sources, `_basesort` donne pour chaque table identifiée par un entier `n` sa sorte et `_instance` donne pour chaque table l'ensemble des n-uplets de celle-ci. On note ici qu'on est dans une sémantique ensembliste donc nous n'avons pas de doublons.

Pour passer des bases de données de type `db_state_` à celle de type `db_state`, on annote ensuite chaque n-uplet des tables sources. Les variables sont ici des couples d'entiers naturels. Le premier élément du couple correspond à l'identifiant d'une table source et le deuxième élément à l'identifiant d'un n-uplet de cette table. Par construction du type `relname_`, chaque table est identifiée par un entier naturel unique. Pour l'attribution des identifiants des n-uplets, on prend le support de la relation source et on range les n-uplets par ordre croissant. On numérote ainsi les n-uplets en partant de 0 pour le plus petit des n-uplets et on augmente de 1 pour chaque n-uplet suivant. Les couples d'attributs associés à un n-uplet d'une table r sont la répétition des attributs de la sorte de r.

La fonction d'annotation d'une table source `Rel n` est donc créée à partir d'une base de données `db` de `db_state_` de la façon suivante :

```
Definition f_inst_ db n t :=
  if Feset.mem t (_instance db n) && (labels t =?_basesort db n)
     && (Oset.mem_bool (Rel n) (_relnames db))
  then create_annot db n t
  else empty SPol.
```

Il faut tout d'abord vérifier que `t` est dans la table source `Rel n`. Si ce n'est pas le cas l'annotation doit être l'ensemble vide. Sinon, nous vérifions des conditions de bonnes formations : la sorte de `t` doit être équivalente à la sorte de la relation et `Rel n` doit être dans les tables source de la base `db`. Si les conditions de bonnes formations ne sont pas vérifiées, l'annotation est l'ensemble vide. Si elles le sont la fonction `create_annot` crée une annotation comme proposé précédemment à partir de la base de données, de l'identifiant de la relation source et du n-uplet.

Le support, lui, filtre les n-uplets de `_instance` pour ne garder que les n-uplets qui ont pour sorte la sorte de la relation.

```
Definition support_inst_ db r :=
  Feset.filter (fun t ⇒ labels t ≡? _basesort db r) (_instance db r).
```

Pour la sorte de la Whr-relation, on garde la sorte de `_basesort`.

Il reste à instancier la fonction `var_to_tuple` qui associe une variable au n-uplet de la table correspondante s'il existe.

```
Definition var_to_tuple_ db (x : N*N) :=
  nth_elt (Feset.elements (support_inst db (Rel (fst x)))) (snd x).
```

On utilise le premier élément de la variable pour trouver la table source puis on cherche l'élément numéro `snd x` dans le support. La fonction `nth_elt` renvoie le n-uplet correspondant si `snd x` ne dépasse pas la taille du support et renvoie `None` sinon.

Avec ces définitions, les hypothèses `well_formed_db`, `well_formed_db1` et `well_formed_db2` et les invariants de `var_to_tuple` sont vérifiés.

Pour obtenir la propriété **(Equivalence-annotations-n-uplets)**, il reste à montrer que :

```
Lemma base_env_eq_inst : ∀ db r x1 x2,
  eq x1 x2 →
  base (f_inst db r x1) =SE= base (f_inst db r x2).
```

ce qui est vrai par la définition de `f_inst` qui est définie à partir de fonctions compatibles avec l'équivalence des n-uplets.

On a donc la propriété **(Equivalence-annotations-n-uplets)** :

```
Theorem equal_tuple_final_ : ∀ db q1 q2 t1 t2,
    well_formed q1 = true →
    well_formed q2 = true →
    is_zero (f (eval_prov_ db nil q1) t1) = false →
    is_zero (f (eval_prov_ db nil q2) t2) = false →
    equal (f (eval_prov_ db nil q1) t1) (f (eval_prov_ db nil q2) t2) = true →
    eq t1 t2.
```

Dans cette instanciation, nous cherchons aussi vérifier l'existence et l'unicité d'une variable pour chaque n-uplet et chaque relation source.

Le théorème qui suit donne : pour tout n-uplet `t` d'une table source `r` de la base de donnée `db` et dont la sorte est équivalente à la sorte de la relation `r`, la base de l'annotation de `t` est constituée par une seule variable.

```
Theorem exists_annot_inst :
  ∀ db r t,
    Oset.mem_bool (Rel r) (_relnames db) = true →
    t ∈ _instance db r →
    labels t ≡ _basesort db r →
    ∃x , Feset.singleton (Var x) =SE= base (f_inst db (Rel r) t).
```

On a donc l'existence de la variable.

Si une variable `x` est présente dans l'annotation d'un n-uplet de la table source `r1` et d'un n-uplet de la table source `r2`, alors les tables sont les mêmes.

```
Theorem unicity_annot_relname :
  ∀ db r1 r2 t1 t2 x,
    x ∈ base (f_inst db r1 t1) →
    x ∈ base (f_inst db r2 t2) →
    Oset.compare r1 r2 = Eq.
```

Si une variable `x` est dans l'annotation d'un n-uplet `t1` d'une table source et d'un n-uplet `t2` d'une table source, alors les deux n-uplets sont les mêmes.

```
Theorem unicity_annot_tuple :
  ∀ db r1 r2 t1 t2 x,
    x ∈ base (f_inst db r1 t1) →
    x ∈ base (f_inst db r2 t2) →
    eq t1 t2.
```

Il y a donc unicité de la variable.

5.5 . Exemples d'application de la where-provenance

Dans cette section, je propose deux exemples d'application de requêtes pour la sémantique concrète de la where-provenance.

Nous partons de la table source suivante :

	Nb_a	**Nb_b**
t_1	1	2
t_2	2	2

Table 5.19 – Rel 0

Avec la formalisation proposée, le n-uplet t_1 est associé à la variable `Var (0,0)` et le n-uplet t_2 à la variable `Var (0,1)`. Les couples attributs qui leur sont associés sont `(nb_a, nb_a)` et `(nb_b, nb_b)`.

Si j'applique sur la table `Rel 0` la requête $\eta_{Nb\_a:=Nb}(Rel\ 0) \cup \eta_{Nb\_b:=Nb}(Rel\ 0)$, le résultat de celle-ci donne :

```
((Var (0, 0), (nb, nb_a) :: nil), (nb, 1)) ::
(((Var (0, 0), (nb, nb_b) :: nil),
          (Var (0, 1), (nb, nb_a):: (nb,nb_b) :: nil)),
           (nb, 2)) :: nil
```

Cela se lit de la manière suivante :

— pour le n-uplet dont la valeur est "1" pour l'attribut "Nb", l'annotation est : la variable `Var (0,0)` associée au couple d'attributs `(nb, nb_a)` (élément en violet)

— pour le n-uplet dont la valeur est "2" pour l'attribut "Nb", l'annotation est : la variable `Var (0,0)` associée au couple d'attributs `(nb, nb_b)` et la variable `Var (0,1)` associée aux deux couples d'attributs `(nb, nb_a)`, `(nb, nb_b)` (élément en orange)

— le reste des n-uplets sont annotés à l'ensemble vide

C'est bien ce qu'on attend. La valeur "1" est héritée de la valeur du n-uplet t_1 contenue dans la colonne "Nb_a". La valeur "2" est héritée de trois cases : la case du n-uplet t_1 contenue dans la colonne "Nb_b", celle du n-uplet t_2 contenue dans la colonne "Nb_a" et celle du n-uplet t_2 contenue dans la colonne "Nb_b".

Pour le deuxième exemple, nous partons des tables sources suivantes :

	Nb_day	**Present**
t_1	2	True

Table 5.20 – Rel 0

	Present
t_2	True
t_3	False

Table 5.21 – Rel 2

Le n-uplet t_1 est associé à la variable `Var (0,0)`, le n-uplet t_2 à la variable `Var (2,0)` et le n-uplet t_3 à la variable `Var (2,1)`. Les couples d'attributs pour t_1 sont "nb_day"^{2} et "present"^{2}. Pour les deux autres n-uplets, le couple d'attributs associé est "present"^{2}.

Lorsqu'on applique la jointure naturelle entre les deux tables, on obtient :

```
(((Var (0, 0), (nb_day, nb_day) :: (present, present) :: nil)
        :: (Var (2, 0), (present, present) :: nil) :: nil),
      (nb_day, 2) :: (present,true) :: nil)
      :: nil
```

Cela signifie que la table résultante contient qu'un seul n-uplet, $(2, true)$. Pour l'attribut "Nb_day", la valeur est héritée du n-uplet t_1 et de la colonne "Nb_day". Pour l'attribut "Present", la valeur est héritée du n-uplet t_1 et de la colonne "Present" mais aussi du n-uplet t_2 et de la colonne "Present". On remarque que la valeur de t_3 n'intervient pas dans le résultat.

5.6 . Discussion

On a pu voir dans ce chapitre la formalisation de la where-provenance pour l'algèbre relationnelle positive et la preuve de la propriété **(Equivalence-annotations-n-uplets)** sur celle-ci qui n'est pas vérifiée par la how-provenance.

5.6.1 . Formalisation de propriétés

Lors de la phase de recherche de cette thèse, la formalisation de la where-provenance et la preuve de la propriété **(Equivalence-annotations-n-uplets)** ne se sont pas effectuées dans l'ordre présenté ici. Je me suis tout d'abord intéressée aux propriétés que je souhaiterais que les annotations possèdent. J'ai ainsi posé la propriété **(Equivalence-annotations-n-uplets)** puis j'ai tenté d'identifier quel type d'annotations la vérifierait. Avec l'assistant de preuve Coq, j'ai essayé de prouver la propriété sur la sémantique de la how-provenance pour déterminer quels éléments en bloquaient la preuve. Le cas de la projection est celui qui m'empêchait de poursuivre la preuve pour les mêmes raisons que la sémantique de la how-provenance ne peut englober celle de la where-provenance. J'ai ensuite cherché à modifier ce type de provenance pour obtenir la structure la plus simple possible qui vérifie la propriété et je me suis aperçue que je retombais sur une structure connue : la where-provenance.

Comme avec la formalisation et la preuve de cette propriété, il serait intéressant d'étudier la provenance du point de vue de propriétés que l'on souhaite qu'elle vérifie et confronter les différents types de provenance connus à ces propriétés. Cela pourrait permettre une meilleure mise en lumière des différences entre ces types de provenance et ainsi mieux les réconcilier sous une structure commune.

5.6.2 . Sémantique multi-ensembliste

On peut remarquer que pour l'algèbre relationnelle positive, la sémantique de la where-provenance ne propose pas ici d'alternative pour une sémantique multi-ensembliste contrairement à la how-provenance. En effet dans la how-provenance, si nous voulons annoter nos tables avec un semi-anneau A et nous placer dans la sémantique multi-ensembliste, il faut annoter les tables avec semi-anneau (A * $\mathbb{N}$). On remarque ici qu'un tel semi-anneau n'est pas une instance de `Integ` si nous prenons la relation d'ordre à laquelle on tendrait naturellement, c'est-à-dire celle telle que :

— $(a, n) <_{A*\mathbb{N}} (b, m)$ si et seulement si $a <_A b$ ou ($a =_A b$ et $n <_{\mathbb{N}} m$).

— $(a, n) =_{A*\mathbb{N}} (b, m)$ si et seulement si $a =_A b$ et $n =_{\mathbb{N}} m$

Il est nécessaire que les éléments $(0_A, n)$ et $(a, 0_{\mathbb{N}})$ soit équivalents. En effet, si un des deux éléments d'un couple est nul, il semble naturel de vouloir que cela signifie que le n-uplet ne soit pas dans la table résultante.

Avec la sémantique de la where-provenance, il n'est pas possible de retenir l'information du nombre d'occurrences : même si on annote les n-uplets sources avec plusieurs variables différentes au départ, on ne peut pas savoir comment interpréter les variables présentes dans l'annotation après l'application d'une requête.

Par exemple dans la **Table 5.23**, si le n-uplet t apparaît deux fois dans la table r_6, on devrait avoir t qui apparaît quatre fois dans la table résultante. Cependant, le n-uplet t a la même annotation x : A^2|y : A^2 avant et après la jointure naturelle.

	A	Annot
t	2	x : A^2 | y : A^2

Table 5.22 – table r_6

	A	Annot
t	2	x : A^2 | y : A^2

Table 5.23 – $r_6 \bowtie r_6$

5.6.3 . Ajout d'expressivité et extraction

Comme pour la how-provenance, une des suites logiques à cette formalisation serait d'augmenter son expressivité de la formalisation avec les expressions d'attributs et les fonctions d'agrégation. Cela demanderait par exemple pour les expressions d'attributs de passer de couples d'attributs à des couples (attributs, expression d'attributs). De même, il serait intéressant de procéder à l'extraction en OCaml de la sémantique de la where-provenance pour obtenir un code formellement vérifié.

5.7 . Comparaison entre la sémantique de la where-provenance et celle de la how-provenance

Étant donné que nous disposons des formalisations de la how-provenance et de la where-provenance pour l'algèbre relationnelle, il est légitime de se demander si la sémantique de la where-provenance peut englober celle de la how-provenance.

La sémantique de la where-provenance encapsule-t-elle celle de la how-provenance ?

Considérons les tables r_1 et r_2 qui pour un type de maladie associe le traitement pour le patient. L'attribut "Maladie" est abrégé en "M" et "Traitement" en "T". Les annotations de la where-provenance sont dans la colonne "Whr" et celles de la how-provenance dans la colonne "Hw". On note (x+y : M^2,T^2) l'annotation (x : M^2,T^2|y : M^2,T^2).

M	T	Whr	Hw
Angine	Doli	x : M^2,T^2	a

Table 5.24 – r_1

M	T	Whr	Hw
Angine	Doli	y : M^2,T^2	b

Table 5.25 – r_2

M	T	Whr	Hw
Angine	Doli	x+y : M^2,T^2	a+b

Table 5.26 – $r_3 = r_1 \cup r_2$

M	T	Whr	Hw
Angine	Doli	x+y : M^2,T^2	a×b

Table 5.27 – $r_4 = r_1 \bowtie r_2$

L'union des tables r_1 et r_2 et la jointure naturelle entre r_1 et r_2 renvoient le même n-uplet. (Angine, Doli) a les mêmes annotations pour la where-provenance

dans les deux cas : x+y : M^2,T^2. Cependant, pour la how-provenance, les annotations du n-uplet (Angine, Doli) sont $a + b$ pour l'union et $a \times b$ pour la jointure naturelle. Dans de nombreux semi-anneaux usuellement utilisés, les opérations $+$ et $\times$ sont différentes. Prenons le semi-anneau des entiers : si nous instancions a et b par 2 et par 14, pour l'union, on obtient l'annotation 16 contre l'annotation 28 pour la jointure naturelle.

La sémantique de la where-provenance n'est pas suffisante pour exprimer toute la richesse de la sémantique de la how-provenance : elle ne permet pas de distinguer la situation où deux n-uplets doivent coexister pour obtenir un n-uplet résultant t de celle où un seul suffit à avoir t dans la table résultante.

La sémantique de la how-provenance et celle de la where-provenance ne peuvent être englobées par aucun des deux types de provenance. L'unification de ces deux types de provenance sous une même théorie est un problème épineux en raison de deux obstacles : la sémantique de la where-provenance ne fait pas de différence entre les opérations effectuées dans le cas de l'union et de la jointure naturelle, et la how-provenance ne tient pas compte de la perte d'informations qui peut survenir lors de la projection sur certains attributs.

Le chapitre suivant se concentre sur la réconciliation entre le monde de la how-provenance et celui de la where-provenance.

6 - W(here-h)ow-provenance

Les deux chapitres précédents se sont focalisés sur deux types de provenance en particulier (la how-provenance et la where-provenance). Dans ce chapitre, je propose un nouveau type de provenance, la Where-how-provenance, qui fait le pont entre les deux types de provenance précédents. Pour faciliter la compréhension, on contractera Where-how-provenance en W(here-h)ow-provenance ou Wow-provenance.

Je décris ici la structure algébrique et la sémantique abstraites de la wow-provenance, qui comme dans les chapitres précédents, peuvent être instanciées dans le but d'obtenir les annotations et la sémantique de la where-provenance ou la how-provenance. Nous restons dans le cadre du chapitre précédent concernant l'expressivité des requêtes : celles-ci sont restreintes à l'expressivité des requêtes de l'algèbre relationnelle positive. Nous conservons cette restriction car nous souhaitons vérifier en Coq que la where-provenance est bien une instanciation de la nouvelle structure et que la formalisation en Coq de celle-ci s'étend seulement à l'algèbre relationnelle positive.

Ce chapitre s'articule comme suit. Je commence tout d'abord par énumérer les points communs et les divergences entre les sémantiques de la how-provenance et de la where-provenance. À partir de ceux-ci, je justifie la façon dont j'ai construit la structure algébrique et la sémantique de la wow-provenance. Je présente ensuite la formalisation abstraite en Coq de la structure algébrique dans laquelle sont plongées les annotations ainsi que la sémantique de la wow-provenance. Je prouve que la sémantique de la wow-provenance vérifie bien les théorèmes généraux et que les annotations de la wow-provenance d'un n-uplet pour requêtes équivalentes sont équivalentes. Enfin, je montre que la where-provenance et la how-provenance constituent bien des instances de la wow-provenance.

6.1 . Intuition

Dans le but de créer une structure algébrique et une sémantique regroupant la how-provenance et la where-provenance, nous nous intéressons tout d'abord de façon informelle aux annotations de chacune des provenances et leur propagation lors de l'application d'une requête. Nous analysons les différences et les analogies entre les deux structures mathématiques utilisées, en particulier leur utilisation pour les différents opérateurs de l'algèbre relationnelle positive mais aussi les éléments remarquables de chaque structure algébrique.
Nous utilisons pour cette comparaison la structure algébrique proposée dans le chapitre précédent pour la formalisation de la where-provenance.

6.1.1 . Équivalence algébrique des requêtes

Avant de rentrer dans les détails de la création de la structure algébrique, je reviens sur une condition que les auteurs de [9] ont suivie dans la confection de la how-provenance : si deux requêtes sont équivalentes algébriquement, tout n-uplet a la même annotation dans les tables résultantes de ces requêtes. Je cherche à conserver autant que possible ce critère dans la construction qui suit.

Je rappelle ici les différentes équivalences algébriques de l'algèbre relationnelle positive qui ont un impact sur la confection de la structure algébrique. On suppose que les requêtes qui suivent sont bien formées.

1. **(*Règle 1*)** Associativité de l'union : $(q_1 \cup q_2) \cup q_3 \equiv q_1 \cup (q_2 \cup q_3)$
2. **(*Règle 2*)** Commutativité de l'union : $q_1 \cup q_2 \equiv q_2 \cup q_1$
3. **(*Règle 3*)** Table vide et l'union : $\varnothing \cup q \equiv q$ où $\varnothing$ est la table vide
4. **(*Règle 4*)** Associativité de la jointure naturelle : $(q_1 \bowtie q_2) \bowtie q_3 \equiv q_1 \bowtie (q_2 \bowtie q_3)$
5. **(*Règle 5*)** Commutativité de la jointure naturelle : $q_1 \bowtie q_2 \equiv q_2 \bowtie q_1$
6. **(*Règle 6*)** Table vide et la jointure : $\varnothing \bowtie q \equiv \varnothing$ où $\varnothing$ est la table vide
7. **(*Règle 7*)** Distributivité de la jointure sur l'union : $q_1 \bowtie (q_2 \cup q_3) \equiv (q_1 \bowtie q_2) \cup (q_1 \bowtie q_3)$
8. **(*Règle 8*)** Projection extérieure : si $A \subseteq B$ alors $\pi_A(\pi_B(q)) \equiv \pi_A(q)$
9. **(*Règle 9*)** Table vide et projection : $\pi_A(\varnothing) \equiv \varnothing$
10. **(*Règle 10*)** Distributivité de la projection sur l'union : $\pi_A(q_1 \cup q_2) \equiv \pi_A(q_1) \cup \pi_A(q_2)$
11. **(*Règle 11*)** Semi-distributivité de la projection sur la jointure naturelle : si $\mathbf{Attr_q}(q_1) \cap \mathbf{Attr_q}(q_2) \subseteq A$ alors $\pi_A(q_1 \bowtie q_2) \equiv \pi_A(q_1) \bowtie \pi_A(q_2)$
12. **(*Règle 12*)** Restriction de la projection sur la sorte de la sous-requête : si $B \supseteq \mathbf{Attr_q}(q)$ alors $\pi_A(q) \equiv \pi_{A \cap B}(q)$
13. **(*Règle 13*)** Double renommage : $\rho_{r_2}(\rho_{r_1}(q)) \equiv \rho_{r_2 or_1}(q)$ où $r_2 or_1$ correspond à la composition des fonctions de renommage r_1 et r_2
14. **(*Règle 14*)** Table vide et renommage : $\rho_r(\varnothing) \equiv \varnothing$
15. **(*Règle 15*)** Distributivité du renommage sur l'union : $\rho_r(q_1 \cup q_2) \equiv \rho_r(q_1) \cup \rho_r(q_2)$
16. **(*Règle 16*)** Distributivité du renommage sur la jointure naturelle : $\rho_r(q_1 \bowtie q_2) \equiv \rho_r(q_1) \bowtie \rho_r(q_2)$
17. **(*Règle 17*)** Restriction du renommage sur la sorte de la sous-requête : si $B \supseteq \mathbf{Attr_q}(q)$ alors $\rho_r(q) \equiv \rho_{r_{|B}}(q)$ où $r_{|B}$ est la restriction de r sur l'intersection de l'ensemble de départ de r et l'ensemble B

Certaines équivalences algébriques ne sont pas énoncées ici car elles n'ajoutent pas de conditions supplémentaires à celles imposées par les équivalences citées.

Algèbre	how-provenance	where-provenance
Ensemble	R (semi-anneau)	$Annot_{whr}$
Zero	0_R	$\varnothing$
One	1_R	$1 : \varnothing$
ϵ	si $t =<>$ alors One sinon Zero	si $t =<>$ alors One sinon Zero
ρ_r	$\bullet$	$\rho\_op(r, \bullet)$
π_A	$\sum \bullet$	$\uplus(\pi\_op(A, \bullet))$
σ_P	si $P(t)$ alors $\bullet$ sinon Zero	si $P(t)$ alors $\bullet$ sinon Zero
$\bowtie$	$\bullet \times_R \bullet$	$\bullet \sqcup \bullet$
$\cup$	$\bullet +_R \bullet$	$\bullet \uplus \bullet$

Table 6.1 – Comparatif entre la how-provenance et la where-provenance

On trouve dans le tableau 6.1 un récapitulatif simplifié des opérations et des éléments remarquables de la how-provenance et la where-provenance. Le n-uplet t est le n-uplet d'intérêt dont on calcule l'annotation. Les points $\bullet$ correspondent aux annotations des n-uplets des sous-requêtes.

Commençons la construction de la structure algébrique et de la sémantique de notre nouvelle structure. Je débute par nommer l'ensemble dans lequel nos annotations sont plongées A et par comparer les deux provenances en m'attardant d'abord sur les éléments qu'elles ont en commun.

Éléments analogues

Je débute par recenser les éléments analogues entre les deux structures : les éléments particuliers de chaque ensemble et les opérateurs ayant une sémantique analogue.

Éléments remarquables

Chacune des provenances a deux éléments remarquables : Zero et One. Ils sont utilisés de façon analogue dans le cas de l'opérateur ϵ et de la sélection. Le Zero indique pour les deux sémantiques qu'un n-uplet n'est pas présent dans une table. Dans la table résultante de la requête ϵ, le n-uplet vide est présent et annoté. Cela implique que les éléments Zero et One sont différents.

Je conserve donc ces deux éléments dans ma nouvelle structure algébrique. Ce sont donc deux éléments distincts de l'ensemble de mes annotations A. Ils sont

notés 0_A et 1_A. Un n-uplet est annoté par 0_A pour une table seulement si ce n-uplet n'est pas dans cette table.

Éléments remarquables	Propriétés
0_A	différent de 1_A caractérise la non appartenance à une table
1_A	différent de 0_A

Table 6.2 – Éléments remarquables hypothétiques

L'opérateur ϵ et la sélection

Après avoir fixé ces éléments remarquables, j'ai donc pu définir les opérations sur la provenance effectuées pour l'opérateur de sélection et du n-uplet vide. Celles-ci sont analogues si on identifie les éléments Zero et One entre eux.

Je fixe de manière informelle la sémantique pour ces deux opérateurs comme suit :

— pour l'opérateur $\epsilon : t \mapsto$ si $t =<>$ alors 1_A sinon 0_A

— pour l'opérateur de la sélection $\sigma_P(q) : t \mapsto$ si $P(t)$ alors **annot**(q,t) sinon 0_A

On note **annot**(q,t), l'annotation de t dans la table associée à la requête q.

L'addition et l'union

En comparant les deux provenances, on remarque que les opérations $+$ et $\uplus$ sont toutes deux utilisées dans les cas de l'union et dans la projection. On note cependant que les éléments additionnés et les éléments sur lesquels on applique $\uplus$ dans le cas de la projection ne sont pas complètement similaires.

Cette analyse me permet d'en déduire que je peux identifier ces deux opérations pour l'union et la projection en une opération dans ma structure algébrique. Je note cette opération $+_A$. Cette opération s'effectue entre deux annotations et renvoie une annotation. L'opération $+_A$ doit donc être une opération de type $A \to A \to A$. Elle est utilisée pour l'opérateur union entre les annotations des deux n-uplets des deux tables des sous-requêtes et pour la projection sur des annotations qui peuvent avoir subi une opération au préalable (dans le cas de la where-provenance).

$+_A$ apparaît donc dans la sémantique des deux opérateurs de la manière suivante pour l'opérateur d'union $q_1 \cup q_2 : t \mapsto$ **annot**(q_1,t) $+_A$ **annot**(q_2,t).
On ne peut pas encore fixer la sémantique pour l'opérateur de projection.

Pour que les annotations soient bien équivalentes quand les requêtes équivalentes, comme l'opérateur union de l'algèbre relationnelle est commutatif (***Règle***

Opération	Opérateur(s)
$+_A$	union projection

Table 6.3 – Opérations hypothétiques

2) et associatif **(*Règle 1*)**, $+_A$ doit l'être aussi dans la structure algébrique.

Comme l'union entre une table vide et une autre table r quelconque est équivalente à la relation r **(*Règle 3*)**, les n-uplets de la table finale et leurs annotations sont donc les mêmes que ceux de la table r. Cela étant valable pour n'importe quel n-uplet et n'importe quelle annotation de ce n-uplet dans la table r, on en déduit que 0_A est donc neutre pour $+_A$.

Opération	Propriétés
$+_A$	associativité commutativité 0_A est neutre avec

Table 6.4 – Opérations hypothétiques

Dans la **Section 4.2**, je précisais qu'un n-uplet n'apparaît pas dans la table résultante d'une union de deux tables s'il n'apparaît dans aucune des tables. Cela se traduit donc dans ma structure par l'ajout de la propriété :

$$\forall x, y \in A, x +_A y = 0_A \Rightarrow x = 0_A = y$$

Dans le tableau 6.5, je propose un récapitulatif de la sémantique proposée grâce aux déductions que nous avons eues jusqu'à présent. Les $??$ indiquent les cas pas encore ou non complètement discutés.

Algèbre	wow-provenance
Ensemble	A
Zero	0_A
One	1_A
ϵ	si $t =<>$ alors 1_A sinon 0_A
ρ_r	$??$
π_A	$\sum_A ??$
σ_P	si $P(t)$ alors $\bullet$ sinon 0_A
$\bowtie$	$??$
$\cup$	$\bullet +_A \bullet$

Table 6.5 – Structure hypothétique (version 1)

Divergences entre les deux types de provenance

La how-provenance et la where-provenance ont une sémantique et une structure algébrique qui divergent au niveau des opérateurs du renommage, de la projection et de la jointure naturelle.

Jointure naturelle

Alors que la sémantique de la where-provenance utilise la même opération pour la jointure naturelle et l'union (à l'exception des cas où l'une des annotations des sous-requêtes est égale à Zero et pas l'autre), la how-provenance utilise deux opérations différentes. Par exemple, le semi-anneau des entiers naturels prend comme opération l'addition pour l'union et la multiplication pour la jointure.

Notre structure A doit prendre en compte cette différence avec une deuxième opération $\times_A$ de type $A \to A \to A$. Celle-ci est utilisée dans le cas de l'opérateur jointure naturelle seulement.
La sémantique de la jointure naturelle $q_1 \bowtie q_2$ est donc :
$t \mapsto \textbf{annot}(q_1, t_{|\textbf{Attr}_\textbf{q}(q_1)}) \times_A \textbf{annot}(q_2, t_{|\textbf{Attr}_\textbf{q}(q_2)})$.

L'opérateur jointure naturelle est associatif (***Règle 4***) et commutatif (***Règle 5***). Pour conserver les équivalences algébriques, $\times_A$ doit être associative et commutative. De même, la jointure naturelle étant distributive (***Règle 7***) sur l'union, $\times_A$ doit être distributive sur $+_A$.

Lors de la jointure d'une table r avec une table vide (***Règle 6***), la table résultante est vide. Comme cela est valide pour tout n-uplet et toute annotation de la table r, on en déduit que l'élément 0_A est absorbant pour $\times_A$.

On pourrait s'attendre à ce que l'élément 1_A soit neutre pour $\times_A$ dans notre structure car c'est le cas pour la how-provenance. Cependant, cette caractéristique n'est pas vérifiée pour la where-provenance donc elle ne fait pas partie des propriétés retenues.

Opération	Propriétés
$\times_A$	associatif commutatif distributif sur $+_A$ 0_A est absorbant sur cette opération

Table 6.6 – Opérations hypothétiques

De plus, comme précisé dans la **Section 4.2**, un n-uplet t n'apparaît pas dans la table résultante d'une jointure de deux sous-requêtes q_1 et q_2 si $t_{\textbf{Attr}_\textbf{q}(q_1)}$ n'est

pas dans la table associée à q_1 ou $t_{\mathbf{Attr_q}(q_2)}$ n'est pas dans la table associée à q_2. La structure doit donc vérifier la propriété :

$$\forall x, y \in A, x \times_A y = 0_A \Rightarrow x = 0_A \vee y = 0_A$$

Renommage et Projection

Étudions en détail la sémantique des deux opérations.

Pour le renommage, l'opération effectuée sur les annotations est la fonction identité dans le cadre de la how-provenance et ρ_op pour la where-provenance.

Pour la projection, on effectue soit des additions sur les annotations soit on applique $\uplus$ selon le type de provenance. Cependant, pour la where-provenance, l'opération π_op est appliquée au préalable sur les annotations.
En observant quelles sont les annotations de la table associée à la sous-requête sur lesquelles on applique ces opérations, nous en déduisons que ces annotations sont associées aux mêmes n-uplets que nous nous plaçons dans la how-provenance ou la where-provenance. Prenons t un n-uplet dans une table résultante. Pour un renommage par r, l'annotation qui subit les opérations pour donner l'annotation de t est celle étant associée au n-uplet $t[r^{-1}]$ dans la table de la sous-requête. Pour une projection sur l'ensemble d'attributs C, les annotations sont celles des n-uplets dont la projection sur C est égale au n-uplet t.

Je regroupe ces deux opérateurs ensemble pour deux raisons. La première étant que dans la formalisation en Coq, ces deux opérateurs sont regroupés en un seul opérateur η. La deuxième est que les opérations (ρ_op et π_op) appliquées sur les annotations pour ces opérateurs dans la where-provenance sont similaires. Le point commun entre ρ_op et π_op est qu'elles agissent sur les couples d'attributs et pas sur l'annotation en entier et ce type d'opération ne correspond à aucune des opérations précédentes.

Je définis donc $\cdot_{A,B}$ qui correspond à l'enchaînement de π_op et ρ_op. Nous remarquons que $r, a \to \rho\_op(r, \pi\_op(Ent(r), a))$ prend en premier argument une fonction de renommage et en deuxième une annotation. Cela nous indique que $\cdot_{A,B}$ n'est pas de type $A \to A \to A$ et qu'il faut introduire un nouvel ensemble que nous notons B qui correspond à l'ensemble des fonctions de renommage pour la where-provenance. Je pose le type de $\cdot_{A,B}$ comme étant $B \to A \to A$. On remarque que l'élément choisi dans B lors de l'application de la requête dépend de l'ensemble sur lequel on souhaite projeter ou de la fonction de renommage que l'on a choisie. Cet élément est donc indexé par l'ensemble d'attributs ou la fonction de renommage.

Pour obtenir une analogie avec la how-provenance, il suffit d'instancier $\cdot_{A,B}$ avec une fonction qui ignore le premier argument et renvoie le deuxième par défaut $x, y \to y$. L'ensemble B peut prendre n'importe quelle forme tant qu'il contient au moins un élément.

Comme l'addition ou l'application de $\uplus$ correspond à l'opération $+_A$, on en déduit ainsi la sémantique suivante :

— pour le renommage $\rho_r(q) : t \rightarrow b_r \cdot_{A,B}$ **annot**(q,t) où b_r est un élément de B

— pour la projection $\pi_C(q) : t \rightarrow \sum_{t'|t=t'_{|C}} b_C \cdot_{A,B}$ **annot**(q,t') où b_C est un élément de B qui dépend de l'ensemble d'attribut C

— pour $\eta_r(q) : t \rightarrow \sum_{t'|t=t'[r]} b_r \cdot_{A,B}$ **annot**(q,t') où b_r est un élément de B qui dépend de la fonction r

Regardons maintenant les propriétés algébriques que doit avoir l'opération $\cdot_{A,B}$.

Propriétés sur les éléments remarquables

Je commence par le comportement de $\cdot_{A,B}$ avec les éléments remarquables 0_A et 1_A :

— Avec ***(Règle 9)*** et ***(Règle 14)***, n'importe quel n-uplet étant annoté par 0_A donne après projection ou renommage un n-uplet ayant pour annotation 0_A. Nous en déduisons que : $\forall b \in B, b \cdot_{A,B} 0_A = 0_A$. De plus, un n-uplet dont la projection (réciproquement le renommage) n'apparaît pas dans la table résultante n'apparaît pas dans la table associée à la sous-requête. On a donc aussi : $\forall b \in B, \forall a \in A, b \cdot_{A,B} a = 0_A \Rightarrow a = 0_A$

— Pour l'opérateur ϵ, la projection sur un ensemble A ou le renommage avec une fonction r laissent inchangé le n-uplet vide et son annotation reste égale à 1_A pour la how-provenance et la where-provenance. Nous avons donc : $\forall b, b \cdot_{A,B} 1_A = 1_A$.

Propriétés de distributivité

Pour ce qui est des propriétés par rapport aux autres opérations, nous avons ce qui suit.

C	D	Ann
t_C	t_D	x : C²,D² $= a_1$

Table 6.7 – relation r_0

C	D	Ann
t_C	t_D	y : C²,D² $= a_2$

Table 6.8 – relation r_1

C	Ann
t_C	$\pi\_op(C,$ x : C²,D²$)\uplus$ $\pi\_op(C,$ y : C²,D²$)$ $= b_C \cdot_{A,B} (a_1 +_A a_2)$

Table 6.9 – $\pi_C(r_0) \cup \pi_C(r_1)$

C	Ann
t_C	$\pi_op(C,$x : C²,D² | y : C²,D²$)$ $= (b_C \cdot_{A,B} a_1) +_A (b_C \cdot_{A,B} a_2)$

Table 6.10 – $\pi_C(r_0 \cup r_1)$

Avec **(*Règle 10*)**, la projection est distributive sur l'union. Les **Tables 6.9 et 6.10** sont donc équivalentes et on souhaite donc avoir les mêmes annotations pour le n-uplet t_C. Le renommage est aussi distributif sur l'union avec **(*Règle 15*)**. Pour avoir ces équivalences au niveau des annotations, il faut que l'opération $\cdot$ soit distributive à droite sur $+_A$:

$$\forall b \in B, (a_1, a_2) \in A^2, b \cdot_{A,B} (a_1 +_A a_2) = (b \cdot_{A,B} a_1) +_A (b \cdot_{A,B} a_2)$$

Les règles **(*Règle 11*)** et **(*Règle 16*)** indiquent que la projection et le renommage sont au moins partiellement distributifs sur la jointure naturelle. J'ajoute donc la distributivité à droite sur $\times_A$ de $\cdot_{A,B}$:

$$\forall b \in B, (a_1, a_2) \in A^2, b \cdot_{A,B} (a_1 \times_A a_2) = (b \cdot_{A,B} a_1) \times_A (b \cdot_{A,B} a_2)$$

Je ne prends pas en compte le critère sur les sortes pour la distributivité de la projection. En effet, l'opération $\sqcup$ et π_op commute dans tous les cas et $\cdot_{A,B}$ ignore le premier élément pour la how-provenance. De plus, comme il est toujours possible d'avoir des tables vérifiant le critère sur les sortes (notamment en prenant des tables dont les sortes sont disjointes) et d'annoter les n-uplets avec n'importe quelle annotation, la propriété doit couvrir tous les éléments de A et de B.

Opération	**Propriétés**
$\cdot_{A,B}$	distributif à droite pour $+_A$ et $\times_A$ 0_A est absorbant à droite sur cette opération 1_A est absorbant à droite sur cette opération

Table 6.11 – Opérations hypothétiques

Propriétés de composition

Pour ce qui est des règles **(*Règle 8*)** et **(*Règle 13*)**, on va s'intéresser ici à l'opérateur η et définir la règle résultante de l'application de deux opérateurs η consécutifs.

Regardons sur des exemples simples ce qu'il se passe lors de l'application de deux fois η. Nous observons que :

1. $\eta_{b:=c}\eta_{a:=b} \equiv \eta_{a:=c}$ car on projette sur l'attribut a qui est renommé en b puis sur l'attribut b qui est renommé en c
2. quand $b \neq c$, l'application de $\eta_{c:=d}\eta_{a:=b}$ renvoie la table vide donc $\eta_{c:=d}\eta_{a:=b} = \eta_\emptyset$.
3. lorsqu'on applique la fonction de renommage vide pour l'opérateur η, on obtient une table vide. On a donc $\eta_r\eta_\emptyset = \eta_\emptyset\eta_r = \eta_\emptyset$

On devine ici qu'une requête appliquant successivement l'opérateur η de deux fois est équivalente à une requête appliquant l'opérateur η une seule fois.

Je définis alors une opération de composition de deux fonctions de renommage. Le résultat de cette opération est la fonction de renommage qu'il faut appliquer pour obtenir le même résultat que si on appliquait les deux fonctions initiales successivement.

Définition 6.1: Composée de deux fonctions de renommage

L'opérateur de composition est noté $\circ$. Soit r et r' des fonctions de renommage, a, b, c et d des attributs. On définit la composée de deux fonctions de renommage inductivement :

- $r \circ \varnothing = \varnothing \circ r = \varnothing$
- $(c := d, r) \circ a := b = \begin{cases} a := d & \text{si } b = c \\ r \circ a := b & \text{sinon} \end{cases}$
- $r \circ (a := b, r') = (r \circ a := b), (r \circ r')$

On remarque que la fonction de renommage résultante est bien formée si les fonctions en entrée le sont.

Avec cette définition et les règles **(*Règle 8*)** et **(*Règle 13*)**, nous en déduisons une nouvelle règle que nous appelons **(*Règle 18*)** :

$$\forall r_1, r_2, \eta_{r_1}\eta_{r_2} \equiv \eta_{r_1 \circ r_2}$$

.

Dans les **Tables 6.13 et 6.14**, j'illustre cette règle. Le n-uplet résultant t_D est le même si on applique les deux fonctions η successivement ou directement la composée des deux fonction de renommage. Pour les annotations, on veut que les annotations de t_D soient égales, donc avoir : $b_{G:=H} \cdot_{A,B} b_{C:=F,D:=G} \cdot_{A,B} a = b_{D:=H} \cdot_{A,B} a$.

C	D	E	Ann
t_C	t_D	t_E	a

Table 6.12 – relation r_3

H	Ann
t_D	$b_{D:=H} \cdot_{A,B} a$

Table 6.14 – $\eta_{D:=H}(r_3)$

H	Ann
t_D	$b_{G:=H} \cdot_{A,B} (b_{C:=F,D:=G} \cdot_{A,B} a)$

Table 6.13 – $\eta_{G:=H}\eta_{C:=F,D:=G}(r_3)$

Pour cela il est nécessaire d'ajouter une opération que l'on note $\odot_B$ de type $B \to B \to B$. Elle doit permettre de faire le pont entre l'annotation $b_{G:=H} \cdot_{A,B}$

$b_{C:=F,D:=G} \cdot_{A,B} a$ et l'annotation $b_{D:=H} \cdot_{A,B} a$. Il faut donc que $b_{G:=H} \odot_B b_{C:=F,D:=G} = b_{D:=H}$ et $b_{G:=H} \cdot_{A,B} (b_{C:=F,D:=G} \cdot_{A,B} a) = (b_{G:=H} \odot_B b_{C:=F,D:=G}) \cdot_{A,B} a$.

De manière plus générale, j'ai besoin d'avoir les deux propriétés suivantes :

1. pour toutes fonctions de renommage r_1, r_2 bien formées, $b_{r_1 \circ r_2} = b_{r_1} \odot_B b_{r_2}$
2. $\forall a \in A, b_1, b_2 \in B^2, b_1 \cdot_{A,B} (b_2 \cdot_{A,B} a) = (b_1 \odot_B b_2) \cdot_{A,B} a$

Dans la how-provenance, l'opération $\odot_B$ peut être instanciée simplement par une fonction constante égale à un élément de B quelconque. Pour la where-provenance, l'ensemble B étant l'ensemble des fonctions de renommage, $\odot_B$ est instanciée vers $\circ$ qui vérifie trivialement les deux propriétés.

Ensemble	Propriétés
B	non vide une opération $\odot_B$

Table 6.15 – Deuxième ensemble hypothétique

Propriétés de restriction

Les règles **(*Règle 17*)** et **(*Règle 12*)** indiquent qu'il est possible de restreindre l'ensemble sur lequel on projette ou la fonction de renommage et d'obtenir la même table résultante selon certaines conditions.

Pour l'opérateur η, cela revient à : si C est un ensemble tel que $C \supseteq \mathbf{Attr_q}(q)$,

$$\begin{aligned} \eta_r(q) &\equiv \rho_r(\pi_{Ent(r)}(q)) \equiv \rho_r(\pi_{Ent(r) \cap C}(q)) \\ &\equiv \rho_r(\pi_{Ent(r \circ id_C)}(q)) \equiv \eta_{r \circ id_C}(q) \end{aligned}$$

car $r_{|C} = r \circ id_C$.

Avec ce qu'on vient de voir précédemment, on a : $\eta_r(q) \equiv \eta_r \eta_{id_C}(q)$ Par ailleurs, on a aussi que si $C \supseteq \mathbf{Attr_q}(q)$, $\eta_{id_C}(q) = q$.

Donc pour un n-uplet t annoté par a et $C \supseteq \mathbf{Attr_q}(q)$, l'annotation $b_C \cdot_{A,B} a$ doit être égale à a.

On aurait envie de mettre donc la propriété suivante : pour tout ensemble d'attributs C, $\forall a \in A, b_{id_C} \cdot a = a$. Cependant dans le cadre de la where-provenance, cette propriété n'est pas valable pour tous les éléments de A.

C	D	Ann
t_C	t_D	x : C^2,D^2

Table 6.16 – relation r_4

F	Ann
t_C	x : C^2

Table 6.17 – $\eta_{id_C}(r_4)$

Le n-uplet résultant de l'application $\eta_{id_C}(r_4)$ sur la **Table 6.16** a pour annotation x : C^2 alors que si la propriété était vérifiée, il devrait garder la même annotation.

J'introduis donc un prédicat P qui prend un élément a de A et un élément b de B et renvoie vrai si la propriété $b \cdot a = a$ est vraie et faux sinon. La propriété est donc

$$\forall a \in A, b \in B, P\ a\ b \Rightarrow b \cdot_{A,B} a = a$$

Il reste maintenant à relier le prédicat P aux règles (***Règle 17***) et (***Règle 12***). Pour cela, j'ajoute que le prédicat P doit vérifier :

$$\forall q, \forall t, P\ annot_q(t)\ b_{id_C} \text{ si } C \supseteq \mathbf{Attr_q}(q)$$

Ce dernier ajout ne pose pas de problèmes à la how-provenance : on peut prendre comme prédicat $P = \top$ comme $\cdot_{A,B}$ ignore l'élément de B.

Pour la where-provenance, on rappelle que l'ensemble B correspond aux fonctions de renommage et que les premiers éléments des couples d'attributs de l'annotation d'un n-uplet t sont dans la sorte de la sous requête (comme nous l'avons vu dans la **Sous-section 5.3.2**). Ainsi un ensemble C d'attributs est inclus dans $\mathbf{Attr_q}(q)$ si et seulement si $Ent(id_C)$ est inclus dans $\mathbf{Attr_q}(q)$. Comme $b_r = r$ dans la where-provenance, on peut donc prendre comme prédicat P le prédicat qui vérifie les premiers éléments des couples d'attributs de l'annotation d'un n-uplet t sont inclus $Ent(b_{id_C})$.

Notre structure évolue comme décrit dans 6.18.

Algèbre	**wow-provenance**
Ensemble	A
Zero	0_A
One	1_A
ϵ	si $t =<>$ alors 1_A sinon 0_A
ρ_r	$b_r \cdot_{A,B} \bullet$
π_{Att}	$\sum_A b_{Att} \cdot_{A,B} \bullet$
σ_P	si $P(t)$ alors $\bullet$ sinon 0_A
$\bowtie$	$\bullet \times_A \bullet$
$\cup$	$\bullet +_A \bullet$

Table 6.18 – Structure hypothétique (version 2)

Cette structure permet de prendre en considération les différents éléments propres de chaque type de provenance mais aussi leurs similitudes.

Maintenant que nous avons notre structure hypothétique, pour obtenir chaque type de provenance, il suffira d'instancier chaque élément particulier et opération de notre nouvelle structure avec les éléments et les opérations correspondants de la structure d'intérêt.

6.1.3 . Instanciation avec la How-provenance

Pour la how-provenance, l'instanciation de la structure hypothétique s'effectue comme présenté dans le tableau 6.19.
Je propose que l'ensemble B soit quelconque avec au moins un élément b et l'opération $\odot_B$ renvoie toujours b. Avec cette instanciation, nous conservons bien

Algebra	Instanciation
A	semi-anneau d'intérêt R
0_A	0_R
1_A	1_R
0_B	b
b_r	b
b_{Att}	b
$\bullet +_A \bullet$	$\bullet +_R \bullet$
$\bullet \times_A \bullet$	$\bullet \times_R \bullet$
$\cdot_{A,B}$	$\_, y \to y$
$\odot_B$	$\_, \_ \to b$

Table 6.19 – Instanciation hypothétique (semi-anneaux)

la distinction entre l'addition et la multiplication.

6.1.4 . Instanciation avec la Where-provenance

Pour la where-provenance, je propose l'instanciation comme présenté dans le tableau 6.20.
L'ensemble B est pour ce type de provenance les fonctions de renommage. Pour

Algebra	Instanciation
A	$Annot_{whr}$
0_A	$\varnothing$
1_A	$1 : \varnothing$
0_B	$\varnothing$
b_r	r
b_{Att}	id_{Att}
$+_A$	$\bullet \uplus \bullet$
$\times_A$	$\bullet \sqcup \bullet$
$\cdot_{A,B}$	$\rho\_op(\bullet, \pi\_op(Ent(\bullet), \bullet))$
$\odot_B$	$r_1, r_2 \to r_1 \circ r_2$

Table 6.20 – Instanciation hypothétique (where-provenance)

chaque projection sur un ensemble d'attributs Att, l'élément b_{Att} est la fonction de renommage d'identité sur Att. Pour le renommage r, l'élément dans b choisi est la fonction de renommage r.

Nous avons donc une structure algébrique et une sémantique hypothétique qui permettent de regrouper sous une même structure la how-provenance et la where-provenance. Dans la section suivante, nous décrivons un cadre formel issu de l'intuition présentée en décrivant la structure algébrique choisie, ses propriétés ainsi que la sémantique de cette nouvelle algèbre.

6.2 . Sémantique

Je reprends ici les éléments particuliers, les opérations et les propriétés sur ces opérations que nous avons déduits dans la **Section 6.1**.

Définition 6.1: Structure Algébrique

Posons A un ensemble et un ensemble B non vide. A possède deux opérations internes $+_A$ et $\times_A$ et deux éléments spéciaux distincts 0_A et 1_A. B possède une opération interne $\odot_B$. $\cdot_{A,B}$ est une opération qui prend à gauche un élément de B, à droite un élément de A et renvoie un élément de A. J'introduis aussi un prédicat P. Le couple A, B doit vérifier les propriétés suivantes :

1. $(A, +_A, 0_A)$ est un monoïde commutatif
2. $\times_A$ est associative, commutative et distributive sur $+_A$
3. 0_A est absorbant pour $\times_A$
4. $\forall b \in B, b \cdot_{A,B} 0_A = 0_A$
5. $\forall b \in B, b \cdot_{A,B} 1_A = 1_A$
6. $\cdot_{A,B}$ est distributive à droite sur $+_A$:
   $\forall b \in B, \forall a_1, a_2 \in A^2, b \cdot_{A,B} (a_1 +_A a_2) = (b \cdot_{A,B} a_1) +_A (b \cdot_{A,B} a_2)$
7. $\cdot_{A,B}$ est distributive à droite sur $\times_A$:
   $\forall b \in B, \forall a_1, a_2 \in A^2, b \cdot_{A,B} (a_1 \times_A a_2) = (b \cdot_{A,B} a_1) \times_A (b \cdot_{A,B} a_2)$
8. $\forall b_1, b_2 \in B^2, \forall a \in A, b_1 \cdot_{A,B} (b_2 \cdot_{A,B} a) = (b_1 \odot b_2) \cdot_{A,B} a.$
9. $\forall b \in B, \forall a \in A, P\ a\ b \Rightarrow b \cdot_{A,B} a = a.$
10. $\forall a_1, a_2 \in A^2$, si $a_1 +_A a_2 = 0_A$ alors $a_1 = 0_A$ et $a_2 = 0_A$
11. $\forall a_1, a_2 \in A^2$, si $a_1 \times_A a_2 = 0_A$ alors $a_1 = 0_A$ ou $a_2 = 0_A$
12. $\forall b \in B, \forall a \in A$, si $b \cdot_{A,B} a = 0_A$ alors $a = 0_A$

Pour que la structure soit complète, il est nécessaire de définir une fonction $\mathcal{B}$ qui à une fonction de renommage associe un élément de B. Cette fonction vérifie les propriétés suivantes :

Propriété 6.1: Propriétés de composition

1. pour toutes fonctions de renommage r_1 et r_2 bien formées,
   $\mathcal{B}(r_1 \circ r_2) = \mathcal{B}_{r_1} \odot_B \mathcal{B}_{r_2}$
2. pour toute requête bien formée q et tout n-uplet t,
   $P\ annot_q(t)\ b_{id_C}$ si $C \supseteq \mathbf{Attr_q}(q)$

Cette structure algébrique est celle dans laquelle sont plongées les annotations. Elle reste abstraite pour permettre dans un deuxième temps une instanciation vers la provenance désirée. On peut y voir le parallèle avec la how-provenance et l'instanciation vers un semi-anneau particulier.

La sémantique de la wow-provenance reprend elle aussi les déductions proposées dans la **Section 6.1**.

Définition 6.2: Sémantique de la wow-provenance

— $[\![r]\!]_{wow} = t \mapsto$ annot (r, t)
où $annot$ est la fonction qui associe une annotation à chaque n-uplet t d'une relation source r
— $[\![\epsilon_{<>}]\!]_{wow} = t \mapsto 1_A$ si $t = <>$ et 0_A sinon
— $[\![\rho_r(q)]\!]_{wow} = t \mapsto \mathcal{B}_r \cdot_{A,B} [\![q]\!]_{wow}(t[r^{-1}])$
où $\mathcal{B}_r$ est l'élément de B associé à la fonction de renommage r
— $[\![\pi_s(q)]\!]_{wow} = t \mapsto \sum^{A}_{t'|t=t'_{|_s}} \mathcal{B}_{id_s} \cdot_{A,B} [\![q]\!]_{wow}(t')$
où $\mathcal{B}_{id_s}$ est l'élément de B associé à la fonction de renommage identité sur l'ensemble d'attributs s
— $[\![\sigma_f(q)]\!]_{wow} = t \mapsto [\![q]\!]_{wow}(t)$ si $eval(f,t) = \top$ et 0_A sinon
où $eval$ évalue la formule logique f sur le n-uplet t
— $[\![q_1 \bowtie q_2]\!]_{wow} = t \mapsto [\![q_1]\!]_{wow}(t_1) \times_A [\![q_2]\!]_{wow}(t_2)$
où t_i est la projection de t sur l'ensemble des attributs associés à q_i
— $[\![q_1 \cup q_2]\!]_{wow} = t \mapsto [\![q_1]\!]_{wow}(t) +_A [\![q_2]\!]_{wow}(t)$

Avec ces deux définitions, nous pouvons remarquer que cette structure algébrique et cette sémantique sont très semblables à celles proposées pour la how-provenance et where-provenance. On retrouve une sémantique quasiment identique à celle de la how-provenance si on omet le cas de la projection. Cela est cohérent car les différences fondamentales entre la how-provenance et la where-provenance se retrouvent sur la projection et la jointure naturelle. La structure algébrique a perdu la neutralité du 1_A pour $\times_A$ et le caractère absorbant du 0_A pour $\times_A$.

Comme précédemment, la formalisation se fera sur l'opérateur η. La sémantique

pour cet opérateur est donc :

$$[\![\eta_r(q)]\!]_{wow} = t \mapsto \sum_{t'|t=t'[r]} \mathcal{B}_r \cdot_{A,B} [\![q]\!]_{wow}(t')$$

où $proj_s(t)$ est la projection sur l'ensemble des attributs s du n-uplet t et $\mathcal{B}_r$ est l'élément de B associé à la fonction d renommage r

6.3 . Formalisation de la wow-provenance

La formalisation abstraite de la structure algébrique se divise en deux parties. Le but de cette première partie est d'élaguer les propriétés de la structure algébrique non nécessaires aux preuves des théorèmes généraux (**Sous-section 4.5.6**). En effet, je n'ai besoin que du caractère de monoïde commutatif de $(A, +_A, 0_A)$ et que les opérations $\times_A$ et $\cdot_{A,B}$ soient compatibles avec l'équivalence de A. Dans un deuxième temps, je formalise la structure algébrique en entier pour vérifier que les annotations issues de deux requêtes algébriquement équivalentes selon les règles énoncées dans la **Sous-section 6.1.1** d'un n-uplet sont équivalentes.

6.3.1 . Structure algébrique minimale

Commençons par formaliser la structure algébrique minimale pour que les théorèmes généraux soient vérifiés. Comme dans les parties précédentes, la structure algébrique que nous avons choisie est mise sous la forme d'un `Record` :

```
Record Alg (S B : Type) (zero one : S) (plus mul : S → S → S)
       (lambda : B → S → S) (OS : Oeset.Rcd S) :=
  mk_SC
    {
      is_CM : CM zero plus OS;
      mul_eq_compat : ∀ s1 s2 s3 s4,
                      eq s1 s2 → eq s3 s4 →
                      eq (mul s1 s3) (mul s2 s4)
      lambda_eq : ∀ s1 s2 l,
                  eq s1 s2 →
                  eq (lambda l s1) (lambda l s2);
    }.
```

Ici l'opération $\cdot_{A,B}$ est notée `lambda` pour éviter toute confusion avec la fonction `dot` qui donne la valeur d'un n-uplet selon l'attribut choisi. De même, pour l'ensemble A qui devient `S`. On remarque que pour les preuves des théorèmes généraux, seule la compatibilité avec les opérations et le caractère de monoïde commutatif de $(S, +, 0_S)$ sont nécessaires. Il n'y a aucune supposition faite sur la structure de l'ensemble `B`.

J'introduis aussi à une fonction `is_zero : S -> bool` qui détermine si une annotation est nulle. Elle est utilisée notamment dans les invariants des relations annotées. La fonction `is_zero` doit vérifier un certain nombre d'invariants :

```
Record Is0 (S B : Type) (OS : Oeset.Rcd S) (zero : S) (plus mul : S → S → S)
       (lambda : B → S → S) (is_zero : S → bool): Type :=
  mk_is0
    {
      is_zero_zero : is_zero zero = true;
      is_zero_plus : ∀ x y, is_zero x = true → is_zero y = true →
                                    is_zero (plus x y) = true;
      is_zero_mul1 : ∀ x y, is_zero x = true →
                                    is_zero (mul x y) = true;
      is_zero_mul2 : ∀ x y, is_zero y = true →
                                    is_zero (mul x y) = true;
      is_zero_lambda : ∀ b x,
        is_zero (lambda b x) = true → is_zero x = true;
    }.
```

L'enregistrement `Is0` assure les propriétés d'intégrité de l'addition, de la multiplication et du $\cdot_{A,B}$.

On peut dès à présent remarquer que j'utilise ici `is_zero` dans les invariants des Wow-relations comme pour la where-provenance et non la fonction `fun x => Oeset.eq_bool x zero` comme pour la how-provenance. Cela permet de ne pas fixer nos preuves à une relation d'ordre particulière lorsqu'une fonction vérifiant moins de conditions qu'une relation d'ordre suffit. Comme nous l'avons vu pour la where-provenance (**Section 5.2**), on faisait déjà cette distinction. L'annotation indiquant que des n-uplets ne sont pas dans une table est l'ensemble vide, ce qui demande de regarder avec la fonction `is_zero` si la base de l'annotation est vide. Cependant, la propriété **(Equivalence-annotation-n-uplets)** n'a pas besoin d'avoir des informations sur la base des annotations mais seulement sur leurs coefficients pour être vérifiée. Cela nous laisse plus de flexibilité au niveau de la relation d'ordre que nous choisissons.

Je reviendrai sur l'utilisation de chacune des propriétés énoncées dans cette sous-section au fur et à mesure qu'elles sont utilisées pour prouver les théorèmes généraux.

6.3.2 . Structure algébrique complète

Dans cette sous-section, je formalise la structure algébrique proposée pour les annotations afin que celles-ci vérifient la propriété **(Equivalence-requête-annotations)** : si deux requêtes q_1 et q_2 sont équivalentes selon les règles présentées dans la **Sous-section 6.1.1** alors les annotations d'un n-uplet dans la table associée à q_1 et à q_2 sont équivalentes.

L'enregistrement `Alg2` reprend les propriétés que l'algèbre relationnelle doit vérifier et qui sont présentées dans la **Section 6.2** :

```
Record Alg2 (S B : Type) (zero one : S) (plus mul : S → S → S)
       (lambda : B → S → S) (plusB : B → B → B) (OS : Oeset.Rcd S)
       (P : S → B → bool) :=
  mk_S2
    {
```

```
    is_alg : Alg zero one plus mul lambda OS;
    one_0_diff_SR : not (eq zero one);
    mul_assoc : ∀ a1 a2 a3,
                  eq (mul a1 (mul a2 a3)) (mul (mul a1 a2) a3);
    mul_comm : ∀ a1 a2, eq (mul a1 a2) (mul a2 a1);
    mul_distr_plus_l : ∀ a1 a2 b,
      eq (mul b (plus a1 a2))
         (plus (mul b a1) (mul b a2));
    mul_0_abs : ∀ a, eq (mul a zero) zero;
    lambda_zero : ∀ b,
      eq (lambda b zero) zero;
    lambda_one : ∀ b, eq (lambda b one) one;
    lambda_distr_plus : ∀ a1 a2 b,
      eq (lambda b (plus a1 a2))
         (plus (lambda b a1) (lambda b a2));
    lambda_distr_mul :
    ∀ b x y,
      eq (lambda b (mul x y)) (mul (lambda b x) (lambda b y));
    lambda_compo :
    ∀ b1 b2 x, eq (lambda b1 (lambda b2 x)) (lambda (plusB b1 b2) x);
    lambda_restr :
    ∀ b x, P x b = true → eq (lambda b x) x;
    is_integ : Integ zero plus mul OS;
    lambda_zero_integ : ∀ b x,
      eq (lambda b x) zero → eq x zero;
  }.
```

Dans cette partie, `is_zero` est instanciée par la fonction de comparaison :

```
Definition is_zero x = Oeset.eq_bool OS x zero.
```

Il faudra donc trouver une nouvelle relation d'ordre pour la where-provenance qui vérifie cette condition lorsqu'on instancie la wow-provenance par la where-provenance. Pour la how-provenance, cette condition est déjà vérifiée. On note de plus qu'avec cette définition, la fonction `is_zero` vérifie les invariants de `IS0`.

La fonction $\mathcal{B}$ qui associe une fonction de renommage à un élément de l'ensemble B est formalisée par l'hypothèse suivante :

```
Hypothesis lambda_pi : list (select T) → B.
```

Je définis aussi la fonction `rond` qui renvoie le résultat de la composition de deux fonctions de renommage. Elle est caractérisée par les deux lemmes suivants :

```
Lemma rond_mem : ∀ e1 e2 l1 l2,
  In (Select_As (e1, e2)) (rond l1 l2) →
  ∃b : attribute,
    In (Select_As (e1, b)) l1 ⋀ In (Select_As (b,e2)) l2.

Lemma rond_mem2 : ∀ e1 e2 l1 l2 b,
    well_formed_pi l1 = true →
    well_formed_pi l2 = true →
    In (Select_As (e1, b)) l1 →
    In (Select_As (b, e2) l2 →
    In (Select_As (e1,e2)) (rond l1 l2).
```

Si on a un renommage de `e1` à `e2` dans la composition des fonctions de renommage `l1` et `l2`, il existe un attribut `b` tel qu'on a un renommage de `e1` à `b` dans `l1` puis un renommage `b` à `e2` dans `l2`. Et inversement, si on a un renommage de `e1` à `b` dans `l1` et un renommage de `b` à `e2` dans `l2` alors on a un renommage de `e1` à `e2` dans la composition des fonctions de renommage `l1` et `l2`. On remarque que le deuxième lemme nécessite d'avoir en hypothèse que `l1` et `l2` sont bien formées.

Avec ces deux fonctions, je formalise les propriétés de composition (**Section 6.2**) avec les deux hypothèses suivantes :

```
Hypothesis lambda_pi_rond : ∀ l1 l2 a,
    well_formed_pi l1 = true →
    eq (lambda (lambda_pi (rond l1 l2)) a)
       (lambda (plusB (lambda_pi l2) (lambda_pi l1)) a).

Hypothesis lambda_pi_pred : ∀ q env t C,
    well_formed q = true →
    well_formed_envp OS env →
    sort_wrel (eval_prov_alg2 env q) ⊆ C →
    P (f (eval_prov_alg2 env q) t) (lambda_pi (id_A C)) = true.
```

L'hypothèse `well_formed_pi` vérifie que la fonction de renommage `l1` est bien formée. Les autres hypothèses de bonne formation sont similaires à celles proposées pour la where-provenance.

Avec l'enregistrement `Alg2` et les hypothèses précédentes, j'ai pu prouver la plupart des propriétés **(Equivalence-requête-annotations)** en Coq.

6.3.3 . Définition des Wow-relations et de la sémantique de la wow-provenance

Je formalise maintenant les Wow-relations et la sémantique de la wow-provenance. J'aborde ici les éléments de formalisation de la wow-provenance qui divergent des précédentes formalisations de la how-provenance et de la where-provenance. Certaines fonctions ne sont pas répétées ici même si elles y sont définies et utilisées de manière similaire aux chapitres précédents.

La structure choisie pour les Wow-relations est similaire à celle des chapitres précédents : un `Record` avec les trois champs `f`, `supp` et `sort_wrel`. La différence réside dans le type de sortie de `f` qui correspond à la nouvelle structure algébrique. On notera `S`, une instance de `Alg2`.

```
Record Wowrel : Type :=
  mk_wr
   {
     f : tupleT → S;
     support : setT;
     sort_wrel : Fset.set (A T);
   }.
```

En ce qui concerne l'enregistrement des invariants vérifiés par les Wow-relations, j'ai repris ceux proposés pour la where-provenance avec la fonction `is_zero`.

```
Record Wowrel_inv (wr : Wrel) : Prop :=
  mk_wri
    {
     finite_support : ∀ t, t ∈? supp wr = false → is_zero (f wr t) = true;
     sort_labels_supp : ∀ t, t ∈ (supp wr) → labels t ≡ sort_wrel wr;
    }.
```

Pour la fonction d'évaluation de la provenance d'une requête, les Wow-relations sont similaires aux K-relations de la how-provenance sauf pour l'opérateur η qui se calque sur celle de la where-provenance. Je ne présenterai ici que la formalisation de ce dernier.

La fonction pour le champ `f` de la Wow-relation associée à l'opérateur `Q_Pi` est la suivante :

```
Definition f_pi wr env b s t' :=
  fold_left (fun x y ⇒ plus (lambda b (f wr y) x)
    (filter
       (fun t ⇒ Oeset.eq_bool t' (projection_prov (env_pt env t) s (sort_wrel wr)))
       (Feset.elements (support wr)))
    zero.
```

Pour calculer l'annotation d'un n-uplet t, les éléments du support de la Wow-relation de la sous-requête sont tout d'abord filtrés selon que leur projection-renommage par `s` est équivalent à t. La fonction `fold_left` applique ensuite sur chaque annotation des n-uplets restant la fonction $b \cdot_{A,B} \bullet$ puis additionne les résultats.

Les fonctions de bonne formation des environnements et des requêtes ainsi que les fonctions d'équivalences entre les environnements sont similaires à celles proposées pour la sémantique de la where-provenance. En ce qui concerne la sorte des requêtes, celle-ci ne change pas par rapport à la whr.

6.3.4 . Preuves des théorèmes généraux

Pour prouver les théorèmes généraux, on doit dans cette formalisation aussi ajouter les hypothèses sur les tables sources :

```
Hypothesis instance_prov_inv : ∀ r, Wowrel_inv (instance_prov r).
Hypothesis instance_support : ∀ r,
    sort_wrel (instance_prov r) = basesort r.
Hypothesis instance_prov_equiv : ∀ r t1 t2,
    eq t1 t2 →
    Oeset.eq_bool OS (f (instance_prov r) t1) (f (instance_prov r) t2) = true.
```

Ces hypothèses pour la fonction `instance_prov` étaient déjà présentes sous des formes similaires dans les chapitres précédents. Cela confirme l'utilité de ces hypothèses dans les preuves des théorèmes généraux.

Ces derniers sont bien vérifiés avec la structure `Alg` et les invariants `Is0`. La structure de monoïde commutatif est utilisée dans les preuves impliquant les fonctions `fold_left` et `fold_left_wacc`. La propriété `mul_eq_compat` est appliquée dans le cadre de la compatibilité de la fonction `f` avec les équivalences des n-uplets

et des environnements. Il en est de même pour `lambda_eq`. Les propriétés sont la fonction `is_zero` permettent l'application des hypothèses d'induction notamment dans la preuve des invariants des Wow-relations.

On obtient donc les théorèmes suivants :

```
Theorem sort_wrel_basesort : ∀ (env :env_prov) q,
    sort q = sort_wrel (eval_prov_alg1 env q).

Theorem sort_wrel_eq : ∀ n q e1 e2,
    sort_wrel (eval_prov_alg1 e1 q) = sort_wrel (eval_prov_alg1 e2 q).
  Proof.

Theorem eval_formula_prov_eq : ∀ s e1 e2,
    equiv_envp e1 e2 →
    eval_formula_prov eval_prov_alg1 e1 s = eval_formula_prov eval_prov_alg1 e2 s.

Theorem Krel_inv_eval : ∀ env q,
    well_formed_envp env →
    Krel_inv (eval_prov_alg1 env q).

Theorem supp_eq : ∀ q e1 e2,
    equiv_envp e1 e2 →
    supp (eval_prov_alg1 e1 q) =SE= supp (eval_prov_alg1 e2 q).

Theorem equal_f_eval_eq : ∀ e1 e2 q t1 t2,
    equiv_envp e1 e2 →
    eq t1 t2 →
    eq (f (eval_prov_alg1 e1 q) t1) (f (eval_prov_alg1 e2 q) t2) = true.
```

6.3.5 . Théorèmes (Equivalence-requête-annotations)

Dans cette section, on vérifie que les annotations d'un n-uplet dans les tables résultantes de deux requêtes équivalentes algébriquement (selon les règles énoncées dans la **Sous-section 6.1.1**) sont bien équivalentes.

Je vais énoncer maintenant certains des théorèmes qu'on nommera théorèmes **(Equivalence-requête-annotations)**. Certains de ces théorèmes nécessitent des hypothèses supplémentaires de bonne formation comme la bonne formation des environnements ou des conditions sur les sortes des requêtes.

J'ai par exemple formalisé le théorème relatif à la règle ***(Règle 2)*** par :

```
Theorem union_comm : ∀ t e q1 q2,
  eq (f (eval_prov_alg2 e (Q_Set Union q1 q2)) t)
     (f (eval_prov_alg2 e (Q_Set Union q2 q1)) t).
```

Pour les règles concernant les tables vides, je propose une version alternative de ces règles. Par exemple, la règle ***(Règle 3)*** devient : si une des deux annotations d'un n-uplet t dans les sous-requêtes est nulle alors l'annotation dans la table résultante de ce n-uplet est nulle. La règle ***(Règle 3)*** est la conséquence du théorème. J'ai donc le théorème :

```
Theorem union_0_neutral : ∀ t e q1 q2,
```

```
    sort_wrel (eval_prov_alg2 e q1) ≡sort_wrel (eval_prov_alg2 e q2) →
    eq (f (eval_prov_alg2 e q2) t) zero →
    eq (f (eval_prov_alg2 e (Q_Union q1 q2)) t) (f (eval_prov_alg2 e q1) t).
```

Il en va de même pour les autres règles avec la table vide.

Les autres règles de la **Sous-section 6.1.1** ont aussi été vérifiées en Coq.

J'ai vérifié aussi les théorèmes, nommés théorèmes d'intégrité, suivants :

— si un n-uplet n'apparaît pas dans la table résultante de l'union de deux requêtes, alors ce n-uplet n'apparaît dans aucune des tables associées aux deux sous-requêtes

— si un n-uplet t n'apparaît pas dans la table résultante de la jointure naturelle de deux requêtes q_1 et q_2, alors soit la projection de t sur la sorte de q_1 n'est pas dans la table associée à q_1 soit la projection de t sur la sorte de q_2 n'est pas dans la table associée à q_2

— si un n-uplet t n'apparaît pas dans la table résultante d'une projection-renommage $\eta_r(q)$, alors aucun n-uplet t' tel que $t'[r] = t$ n'apparaît dans la table associée à q

Ces théorèmes découlent des propriétés `is_integ` et `lambda_zero_integ`.

On a ainsi les théorèmes :

```
Theorem union_0_integ : ∀ t e q1 q2,
    sort_wrel (eval_prov_alg2 e q1) ≡sort_wrel (eval_prov_alg2 e q2) →
    eq (f (eval_prov_alg2 e (Q_Union q1 q2)) t) zero →
    eq (f (eval_prov_alg2 e q1) t) zero ⋀ eq (f (eval_prov_alg2 e q2) t) zero.

Theorem nj_integ : ∀ t e q1 q2,
    labels t ≡sort_nj (eval_prov_alg2 e q1) (eval_prov_alg2 e q2) →
    eq (f (eval_prov_alg2 e (Q_NaturalJoin q1 q2)) t) zero →
    eq (f (eval_prov_alg2 e q1)
          (mk_tuple (sort_wrel (eval_prov_alg2 e q1)) (dot t))) zero
    ⋁ eq (f (eval_prov_alg2 e q2)
          (mk_tuple (sort_wrel (eval_prov_alg2 e q2)) (dot t))) zero.

Theorem pi_integ : ∀ t e s q,
    well_formed_envp e →
    eq (f (eval_prov_alg2 e (Q_Pi s q)) t) zero →
    ∀ t',
      eq t (projection_prov (env_pt e t') s (sort_wrel (eval_prov_alg2 e q))) = true →
      eq (f (eval_prov_alg2 e q) t') zero.
```

6.4 . Instanciation par un type de provenance

Pour valider que la sémantique de la wow-provenance est un cadre générique englobant à la fois la sémantique de la how-provenance et celle de la where-provenance, il reste à instancier la structure algébrique `Alg2` avec les structures énoncées précédemment pour la how-provenance et la where-provenance et de vérifier qu'avec ces structures algébriques la fonction d'évaluation des requêtes de la

wow-provenance donne des résultats équivalents à ceux de la how-provenance et de la where-provenance. On utilise pour cette section les sémantiques abstraites de la how-provenance (**Section 4.5**) et de la where-provenance (**Section 5.2**) comme instanciation.

6.4.1 . Instanciation de la structure algébrique

Pour instancier la structure algébrique `Alg2` avec les éléments de la how-provenance, on a besoin de poser en hypothèse un semi-anneau commutatif `(K,zero,one,plus,mul,OK)` qui est aussi l'instance `Integ`. On remarque ici que le caractère intègre du semi-anneau n'avait pas été nécessaire précédemment pour les preuves que nous avons faites pour la how-provenance. Cependant, sans l'intégrité de l'addition et de la multiplication, la condition "les n-uplets annotés par `zero` ne sont pas dans la table" ne peut être vérifiée.

Il convient aussi de définir les fonctions $\cdot_{A,B}$, $\mathcal{B}$ et `is_zero`, comme suit :

```
Definition lambda (x y : K) := y.
Definition lambda_pi _:= zero.
Definition is_zero x := Oeset.eq_bool x zero.
```

Ces fonctions correspondent à celles proposées précédemment 6.1.3.

La structure `Alg2` est donc instanciée comme suit :

```
Theorem is_Alg2 :
  Alg2 zero one plus mul lambda (fun __⇒ zero) OK (fun __⇒ true).
```

On remarque que l'opération $\odot_B$ est ici la fonction constante égale à `zero` et le prédicat renvoie toujours vrai. Cette preuve ne pose pas de difficultés particulière. Les propriétés de composition sont aussi trivialement prouvées.

L'instanciation de la structure algébrique `Alg2` avec les annotations de la where-provenance nécessite, elle, de définir une relation d'ordre qui coïncide avec la fonction `is_zero` de la where-provenance en `zero`.

On suppose ici que l'ensemble des variables sont abstraites et on définit la fonction de comparaison :

```
Definition compare_alt x y :=
  match Feset.compare (base x) (base y) with
  |Eq ⇒ Femap3.compare PolK x y
  |x ⇒ x
  end.
```

On compare tout d'abord les bases (c'est-à-dire les ensembles de variables et 1) entre elles puis si celles-ci sont équivalentes on compare les coefficients (les ensembles de couples d'attributs). À partir de cette fonction de comparaison, on définit la relation d'ordre `OWr`. Avec celle-ci, on a bien que :
`is_zero x = Oeset.eq_bool OWr x zero.`

La fonction `renaming_coeff` correspond à la fonction $\mathcal{B}$ et la fonction `fun s1 s2 => Fset_concat s2 s1` à la fonction $\odot_B$. Il reste donc à définir les fonctions $\cdot_{A,B}$ ainsi que le prédicat. La fonction $\cdot_{A,B}$ est définie par :

```
Definition lambda_ s x := Femap3.transpo s x.
```

et le prédicat `P` est défini :

```
Definition P p (b : Fset.set (FAA T)) :=
  (Fset.for_all (fun a ⇒ Oset.eq_bool (OAtt T) (fst a) (snd a)) b) &&
  (Fset.for_all (fun a ⇒ (a,a) ∈? b) (all_attribut_p p)).
```

`p` est une annotation et `b` correspond à la fonction de renommage. La fonction de renommage est ici un ensemble de couples d'attributs. `all_attribut_p` renvoie l'ensemble des premiers éléments des couples d'attributs contenus dans l'annotation `p`. Pour chaque attribut a dans cet ensemble, (a, a) est inclus dans `b`. `P` a aussi besoin de vérifier que la fonction de renommage est la fonction d'identité, c'est-à-dire que chaque premier et second élément des couples d'attributs sont identiques.

Pour rappel, la fonction $\sqcup$ est définie comme suit :

```
Definition mul a1 a2 :=
            if negb (is_zero a1) && negb (is_zero a2)
            then union a1 a2 else zero.
```

On vérifie que les annotations sont non nulles.

On obtient bien que la structure `Alg2` est instanciée par les annotations de la where-provenance :

```
Theorem is_Alg2_ :
  Alg2 empty (singleton One Fset.empty) union mul lambda_
       (fun s1 s2 ⇒ @Fset_concat T s2 s1) OWr P.
```

Pour ce qui est des propriétés de compositions, la première condition ne nécessite pas d'hypothèses supplémentaires en plus des hypothèses de bonnes formations.

```
Lemma lambda_pi_rond : ∀ l1 l2 a,
    well_formed_pi l1 = true →
    well_formed_pi l2 = true →
    eq OWr (lambda_ (lambda_pi_ (rond l1 l2)) a)
       (lambda_ (Fset_concat (lambda_pi_ l1) (lambda_pi_ l2)) a).
```

La seconde condition nécessite de supposer les hypothèses `inst_annot1` et `inst_annot2` sur les relations sources comme proposées dans la **Sous-section 5.2.6**. Sans cela, nous n'avons aucune information sur les couples d'attributs des annotations des tables sources.

```
Lemma lambda_pi_pred_ : ∀ q env t C,
    well_formed q = true →
    well_formed_envp env →
    Fset.subset (sort_wrel (eval_prov__ env q)) C = true →
    P (f (eval_prov__ env q) t) (lambda_pi_ (id_A C)) = true.
```

6.4.2 . Equivalence pour l'évaluation des requêtes avec la wow-provenance

Nous nous intéressons maintenant aux preuves de l'équivalence entre les différents champs des Wow-relations instanciées et ceux de la how-provenance et la where-provenance lors de l'application d'une requête.

Une conséquence directe des théorèmes généraux vérifiés par la how-provenance, la where-provenance et la wow-provenance est que les sortes des K-relations et les Whr-relations obtenues après l'application d'une requête coïncident avec celles des Wow-relations.

```
Theorem eval_prov_sort : ∀ env1 env2 q,
    sort_wrel (eval_prov_wow env1 q) = sort_krel (eval_prov_sr env2 q).

Theorem eval_prov_sort_ : ∀ env1 env2 q,
     sort_wrel (eval_prov_wow env1 q) = sort_whrel (eval_prov_whr env2 q).
```

Pour la how-provenance, les preuves montrant que les fonctions d'annotations (réciproquement les supports) des K-relations coïncident avec celles de la wow-provenance, se font par induction sur la taille de la requête et ne posent pas de difficultés particulières.

```
Theorem sr_eval_prov_f_eq :
  ∀ q t,
    well_formed q = true →
    eq (f (eval_prov_wow nil q) t) (f (eval_prov_sr nil q) t).

Theorem support_eq_hw : ∀ env q,
    well_formed q = true →
    well_formed_envp env →
    support (eval_prov_wow env q) =SE= support (eval_prov_sr env q).
```

Pour la where-provenance, nous avons besoin de définir une seconde fonction de comparaison `compare_alt2` pour définir une deuxième relation d'ordre `OWr2`.

```
Definition compare_alt2 x y :=
  match Oeset.compare OPolK (suppr_one_in_pol x) (suppr_one_in_pol y) with
  |Eq ⇒
     match (One ∈? base x, One ∈? base y) with
     |(true, true) ⇒ Eq
     |(true, false) ⇒ Gt
     |(false, true) ⇒ Lt
     |(false, false) ⇒ Eq
     end
  |x ⇒ x
  end.
```

On compare tout d'abord les coefficients associés aux variables présentes dans les bases (on exclut avec la fonction `suppr_one_in_pol` que les coefficients associés au 1 doivent être équivalents) entre eux puis si ceux-ci sont équivalents, on regarde si l'élément `One` est dans les bases des annotations.

Cette relation d'ordre permet de définir une équivalence entre les environnements :

```
Definition equiv_env_slicep_alt (e1 e2 : group_by * Wowrel) :=
let (g1, kr1) := e1 in
```

```
let (g2, kr2) := e2 in
    sort_wrel kr1 ≡sort_wrel kr2 ⋀ g1 = g2 ⋀
    support kr1 =SE= support kr2 ⋀
    filter_zero kr1 =SE= filter_zero kr2 ⋀
    ∀ x1 x2, eq x1 x2 → eq OWr2 (f kr1 x1) (f kr2 x2).
```

La fonction `equiv_env_slicep_alt` est similaire à la fonction `equiv_env_slicep` de la how-provenance et de la where-provenance. Les fonctions des annotations sont équivalentes selon la relation d'ordre `OWr2` Si on l'a remplacé par la relation d'ordre `Owr`, les preuves d'équivalence entre les champs des Whr-relations et des Wow-relations ne passent pas. Elles imposent trop de contraintes sur les environnements.

Avec cette équivalence, on obtient donc que les fonctions d'annotations et les supports des Whr-relations coïncident avec ceux des Wow-relations et les preuves se font par induction sur la taille de la requête.

```
Theorem whr_eval_prov_support_eq : ∀ q,
    well_formed_ q = true →
    support (eval_prov__ nil q) =SE= support (eval_prov_whr nil q).

Theorem whr_eval_prov_base_f_eq : ∀ q x,
    well_formed_ q = true →
    base (f (eval_prov__ nil q) x) =SE=
      base (f (eval_prov_whr nil q) x).

Theorem whr_eval_prov_coeff_f_eq : ∀ q x,
    well_formed_ q = true →
    eq OPolK (f (eval_prov__ nil q) x) (f (eval_prov_whr nil q) x).
```

On a bien donc que les sémantiques de la how-provenance et la where-provenance coïncident avec les deux instances de la wow-provenance proposée ici. De plus, ces théorèmes, l'instanciation de `Alg2` et la vérification des propriétés de composition confirment que la sémantique de la where-provenance et de la how-provenance vérifient les théorèmes généraux et la plupart des théorèmes (Equivalence-requête-annotations).

6.5 . Instanciation concrète

Pour l'instanciation concrète j'ai repris les mêmes structures que les **Chapitres 4 et 5** et j'ai vérifié que chaque champ des K-relations et Whr-relations après l'application d'une requête sont bien équivalents aux champs des Wow-relations pour cette même requête. Comme je me place dans l'algèbre relationnelle positive, pour la how-provenance, j'ai effectué l'instanciation en deux temps : un premier cas avec un semi-anneau quelconque puis un second avec comme semi-anneau celui des entiers naturels.

6.6 . Discussion

La wow-provenance propose ainsi un cadre structurel et une sémantique permettant de jongler entre la sémantique de la how-provenance et de la where-provenance. Elle joue un rôle unificateur entre les deux structures algébrique mais aussi leur sémantique.

6.6.1 . Vers une provenance hybride entre la how-provenance et la where-provenance

Une perspective directe à cette formalisation est de s'intéresser à une instanciation des Wow-relations avec une structure mixte entre la how-provenance et la where-provenance. Comme remarqué dans la discussion (**Section 5.6**), la sémantique proposée pour la where-provenance ne permet pas de prendre en compte d'éventuels doublons de n-uplets et donc pas de sémantique multi-ensembliste. Il serait intéressant d'étudier le cas où les annotations sont plongées dans le produit cartésien entre l'ensemble des entiers naturels et l'ensemble des annotations de la where-provenance et d'observer si cela correspondrait à une sémantique multi-ensembliste de la where-provenance.

6.6.2 . Expressivité et extraction

Comme pour les autres sémantiques, cette sémantique est sujette à être étendue notamment aux expressions d'attributs et aux fonctions d'agrégations et à être extraite vers Ocaml.

7 - Conclusion et perspectives

Ma thèse représente une avancée significative dans le domaine de la provenance des données, en introduisant un nouveau type de provenance nommé wow-provenance qui unifie la how-provenance et la where-provenance. La wow-provenance apporte une réponse à la communauté des bases de données concernant la délimitation entre la how-provenance et la where-provenance, aucune des deux sémantiques n'incluant l'autre. Il a été nécessaire de construire une structure algébrique novatrice pour les annotations et une sémantique unificatrice pour réunir ces deux types de provenance dans le cadre de l'algèbre relationnelle positive.

Mes contributions sont multiples et se sont articulées entre la formalisation abstraite et concrète des sémantiques de la how-provenance, de la where-provenance et de la wow-provenance. Les requêtes étudiées sont celles de l'algèbre relationnelle positive pour les trois types de provenance. Pour la how-provenance, l'expressivité des requêtes a parfois été élargie aux expressions d'attributs et aux fonctions d'agrégation.

Ces formalisations vérifient à l'aide de l'assistant de preuves Coq les cadres mathématiques proposés pour la how-provenance et la where-provenance. Grâce à Coq, j'ai pu notamment affiner le cadre mathématique concernant les annotations de la how-provenance en ajoutant que les semi-anneaux dans lesquels sont plongées les annotations doivent être intègres pour que les annotations nulles caractérisent l'absence d'un n-uplet dans une table. Plusieurs instances concrètes de semi-anneaux commutatifs et de semi-modules commutatifs ont été formalisées, dont les entiers naturels ou les polynômes à coefficients dans les entiers naturels et à plusieurs indéterminées.

L'utilisation de l'assistant de preuve Coq offre des garanties fortes sur les preuves effectuées et permet de s'assurer que les structures choisies pour formaliser les différentes sémantiques sont bien compatibles avec les équivalences sur les n-uplets et les environnements. Il confirme aussi que les relations annotées issues de l'évaluation des requêtes vérifient bien leurs invariants ainsi que tous les autres théorèmes généraux.

Enfin, j'ai également prouvé, pour chaque type de provenance, une caractéristique à l'aide de l'assistant de preuve. Le théorème d'adéquation montre que la sémantique de l'algèbre relationnelle positive étendue coïncide avec celle de la how-provenance lorsque les annotations sont plongées dans le semi-anneau des entiers naturels. Avec la where-provenance, la propriété **(Propriété équivalence-annotation-n-uplets)** est vérifiée c'est-à-dire que deux n-uplets issus de deux tables résultantes et ayant les mêmes annotations sont équivalents. Pour conclure, la wow-provenance assure que dans la plupart des cas, les tables résultantes de deux requêtes équivalentes ont pour un n-uplet des annotations équivalentes (Equivalence-requêtes-annotation).

Mes travaux de recherche ont ainsi permis de proposer une solution novatrice pour unifier la how-provenance et de la where-provenance et ont été validés à l'aide d'une méthodologie rigoureuse grâce à l'assistant de preuve Coq.

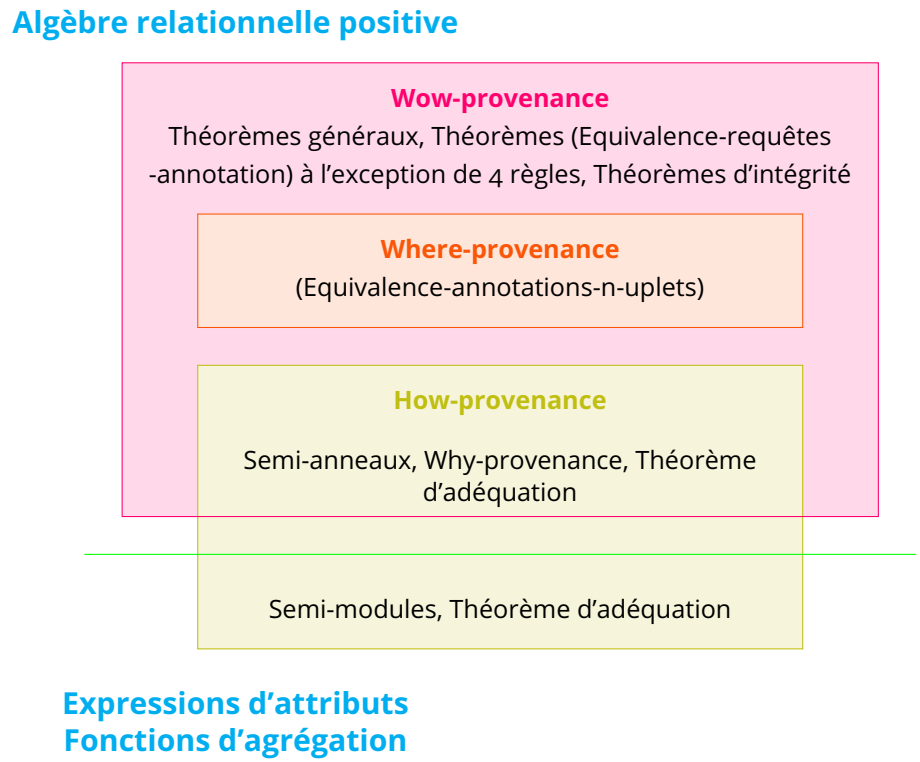


Figure 7.1 – Formalisation en Coq

Perspectives de recherche

Perspectives à court terme

Comme énoncé dans les différentes sections de discussion, plusieurs pistes d'amélioration sont évisagées pour les formalisations de la how-provenance, de la where-provenance et de la wow-provenance. Tout d'abord,il serait intéressant d'étendre l'expressivité des requêtes aux expressions d'attributs, aux fonctions d'agrégation et permettre l'utilisation du "HAVING" dans une requête. Ensuite, une autre piste serait d'extraire en OCaml les formalisations en Coq pour obtenir un code formellement vérifié pour chacun des types de provenance. Cela permettrait de tester ces formalisations sur des cas concrets et d'avoir un modèle de référence auquel d'autres sytèmes d'annotations de données pourraient être confrontés pour détecter des erreurs ou bugs dans les calculs de la provenance des données.

Une autre perspective directe serait de s'intéresser à une instanciation de la wow-provenance avec une structure mixte entre la how- provenance et la where-provenance et d'observer si cela permet d'obtenir une sémantique multi-ensembliste de la where-provenance.

Perspectives à long terme

Cette thèse qui était initialement orientée pour s'appliquer sur des données biologiques, pourrait bénéficier d'une extension de la sémantique de how-provenance et de where-provenance pour inclure les workflows scientifiques annotés. Un workflow est un traitement de données qui se fait par le biais de "pipelines" d'analyses : plusieurs étapes d'analyse de données s'enchaînent et la sortie de chaque étape correspond à l'entrée de la suivante. Ces workflows permettent par exemple de nettoyer, de transformer et d'analyser des données et trouvent des applications dans des domaines tels que la biologie, la physique-chimie ou encore la géologie. Cependant les workflows scientifiques utilisent souvent des outils et des logiciels tiers dont les opérations sont parfois obscures pour l'utilisateur. Ces étapes du processus peuvent être considérées comme des "boîtes noires", ce qui rend difficile la compréhension de l'origine des données et de leur transformation tout au long du flux de travail. En effet, ces boîtes noires peuvent prendre et renvoyer des données sans que l'on connaisse la transformation effectuée sur celles-ci ou quelles données sont utilisées pour obtenir une donnée résultante particulière. Dans ce contexte, l'extension de la sémantique de how-provenance et de where-provenance pourrait être utile pour faciliter la compréhension de la provenance des données dans les workflows scientifiques annotés.

Actuellement pour la provenance dans les workflows scientifiques, chaque donnée issue d'une étape de traitement est annotée avec l'ensemble des annotations des données entrantes, du nom du traitement ainsi que les paramètres de ce traitement. Dans l'article [34], une proposition plus fine est faite pour le langage de requête Pig Latin : il est possible pour certaines étapes de traitement de "zoomer" pour étudier la provenance plus en détail lorsque ces traitements utilisent des opérateurs de l'algèbre relationnelle.

Ma proposition pour étendre la sémantique de how-provenance et de where-provenance aux workflows scientifiques consisterait à inclure non seulement l'algèbre relationnelle étendue, mais aussi les transformations de données dont les caractéristiques sont partiellement connues. Pour atteindre cet objectif, il serait nécessaire de développer des annotations qui reflètent le degré d'incertitude associé à la nature des opérations effectuées sur les données.

La principale différence entre la provenance que nous avons pour l'algèbre relationnelle et celle que je proposerais pour les workflows est que, pour ces derniers, il arrive parfois que nous ne disposions d'aucune information sur la manière dont les données ont été transformées. Cette incertitude doit être prise en compte dans notre proposition de provenance.

Afin de pallier à ce problème, nous pourrions introduire deux nouvelles opé-

rations, $\boxtimes$ et $\circledast$, tout en conservant les éléments particuliers et les opérations spécifiques de l'algèbre de la wow-provenance.

Comme nous l'avons vu précédemment, l'opération $+$ indique que la présence d'une des données dont les annotations sont prises en argument de $+$ est suffisante pour obtenir une donnée résultante particulière, tandis que l'opération $\times$ nécessite la présence des deux données. Pour introduire une notion d'incertitude, l'opération $\boxtimes$ indiquerait que la présence des deux données assure la présence d'une donnée résultante mais nous ne savons pas si une seule des deux données est suffisante, ou si les deux doivent être présentes simultanément. L'opération $\circledast$ indiquerait que les deux données pourraient jouer un rôle dans la production de la donnée résultante, mais que nous ne pouvons pas en être certain. L'annotation zéro serait neutre pour $\circledast$ et permettrait d'indiquer qu'une donnée particulière pourrait jouer ou non un rôle dans la production de la donnée résultante.

Un tel travail de recherche serait bénéfique pour la communauté scientifique en permettant de mieux comprendre la provenance des données dans des contextes complexes où l'algèbre relationnelle étendue ne suffit plus et en fournissant des outils pour faciliter la compréhension et l'analyse des workflows scientifiques.

Bibliographie

[1] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016 : Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[2] Belinda Giardine, Cathy Riemer, Ross C Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, et al. Galaxy : a platform for interactive large-scale genome analysis. *Genome research*, 15(10) :1451–1455, 2005.

[3] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1) :1–9, 2016.

[4] Carole Goble, Sarah Cohen-Boulakia, Stian Soiland-Reyes, Daniel Garijo, Yolanda Gil, Michael R Crusoe, Kristian Peters, and Daniel Schober. Fair computational workflows. *Data Intelligence*, 2(1-2) :108–121, 2020.

[5] Véronique Benzaken, Sarah Cohen-Boulakia, Évelyne Contejean, Chantal Keller, and Rébecca Zucchini. A coq formalization of data provenance. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 152–162, 2021.

[6] Y Richard Wang, Stuart E Madnick, et al. A polygen model for heterogeneous database systems : The source tagging perspective. 1990.

[7] Yingwei Cui, Jennifer Widom, and Janet L Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2) :179–227, 2000.

[8] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where : A characterization of data provenance. In *International conference on database theory*, pages 316–330. Springer, 2001.

[9] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, 2007.

[10] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 153–164, 2011.

[11] Jennifer Widom. Trio : A system for integrated management of data, accuracy, and lineage. Technical report, Stanford InfoLab, 2004.

[12] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. Provsql : Provenance and probability management in postgresql. *Proceedings of the VLDB Endowment (PVLDB)*, 11(12) :2034–2037, 2018.

[13] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. Dbnotes : a post-it system for relational databases based on provenance. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 942–944, 2005.

[14] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. Provenance in databases : Why, how, and where. *Foundations and Trends® in Databases*, 1(4) :379–474, 2009.

[15] Cristina Sirangelo. *Positive Relational Algebra*, pages 2124–2125. Springer US, Boston, MA, 2009.

[16] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings 13th International Conference on Data Engineering*, pages 91–102. IEEE, 1997.

[17] Carlos Scheidegger, David Koop, Emanuele Santos, Huy Vo, Steven Callahan, Juliana Freire, and Claudio Silva. Tackling the provenance challenge one layer at a time. *Concurrency and Computation : Practice and Experience*, 20(5) :473–483, 2008.

[18] Susan B Davidson and Juliana Freire. Provenance and scientific workflows : challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1345–1350, 2008.

[19] Adriane Chapman and HV Jagadish. Why not ? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 523–534, 2009.

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

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

[22] Carlos Gonzalia. Towards a formalisation of relational database theory in constructive type theory. In *International Conference on Relational Methods in Computer Science*, pages 137–148. Springer, 2003.

[23] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. A coq formalization of the relational data model. In *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 23*, pages 189–208. Springer, 2014.

[24] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *ACM Int. Conf. POPL*, 2010.

[25] Joshua S Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1555–1569, 2017.

[26] Véronique Benzaken and Evelyne Contejean. 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*, pages 249–261, 2019.

[27] Véronique Benzaken, Evelyne Contejean, Ch Keller, and Eunice Martins. A coq formalisation of sql's execution engines. In *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 9*, pages 88–107. Springer, 2018.

[28] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql : Proving query rewrites with univalent sql semantics. *ACM SIGPLAN Notices*, 52(6) :510–524, 2017.

[29] The Mathematical Components team. Mathematical Components. Available at https://math-comp.github.io/math-comp.

[30] Évelyne Contejean. Coccinelle, a Coq library for rewriting. In *Types*, Torino, Italy, March 2008.

[31] Yael Amsterdamer, Daniel Deutch, and Val Tannen. On the limitations of provenance for queries with difference. In *3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP 11)*, 2011.

[32] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. Provsql : Gestion de provenance et de probabilités dans postgresql. In *Proc. BDA*, Bucharest, Romania, November 2018. Conference without formal proceedings. (Demonstration).

[33] James Cheney, Anthony Finkelstein, Bertram Ludäscher, and Stijn Vansummeren. Principles of provenance (dagstuhl seminar 12091). In *Dagstuhl Reports*, volume 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[34] Yael Amsterdamer, Susan B Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig : Enabling database-style workflow provenance. *arXiv preprint arXiv :1201.0231*, 2011.