

# Prototyper un compilateur de requêtes avec Coq

---

Joshua Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar et Jérôme Siméon

*IBM Research, USA*

## Résumé

La spécification et le prototypage de nouvelles fonctionnalités sont importantes dans de nombreux projets industriels. Une approche fonctionnelle et l’usage d’outils de vérification formelle peuvent s’avérer particulièrement utiles dans ce contexte, mais créent des défis supplémentaires lors de l’intégration dans un code produit existant ou pendant le transfert de technologie.

Dans cet article, nous présentons comment nous avons utilisé Coq pour le prototypage d’un compilateur de requêtes intégré au produit ODM Insights d’IBM. Nous discutons des avantages et inconvénients de Coq dans ce contexte et décrivons la méthodologie employée pour la ré-implantation en Java nécessaire à l’intégration dans le produit.

## 1. Introduction

Cet article présente notre utilisation de Coq [24] pour le prototypage d’une nouvelle fonctionnalité dans un logiciel commercial. Il s’agit d’un compilateur de requêtes vers la base de données distribuée Cloudant pour *Operational Decision Manager: Decision Server Insights* (ODM Insights) d’IBM réalisé dans le cadre du projet *Middleware for Events, Transactions, and Analytics* [1].

Bien que le but n’étant pas l’obtention d’un compilateur certifié mais d’un prototype en Java à délivrer à l’équipe produit, nous avons décidé d’utiliser Coq comme outil de développement à l’intérieur de l’équipe de recherche. La première raison de ce choix est que Coq est un langage de programmation fonctionnel avec filtrage de motifs et donc adapté à l’écriture de compilateurs. La deuxième raison est la possibilité de faire des preuves. Nous pensions que cela allait nous aider à construire le compilateur. Nous avons identifié en particulier deux aspects qui nous semblaient bénéficier du support de méthodes formelles : (1) la grande différence sémantique entre le langage source (à base de règles sur des objets) et le langage cible (map/reduce sur des documents JSON) et (2) l’optimiseur de requêtes qui est toujours une partie subtile. Enfin, nous étions aussi intéressés par les aspects de recherche que soulevait la réalisation d’un tel compilateur en Coq.

D’autres systèmes formels permettant à la fois la programmation et la vérification auraient pu être utilisés. Coq a été choisi premièrement pour des raisons de familiarité. Au début du projet, un des membres de l’équipe avait une grande expertise de Coq et d’autres membres étaient déjà familiers avec OCaml et souhaitent apprendre Coq. L’existence de projets sur la vérification de systèmes de gestion de données [21, 3] nous a également rassuré sur la maturité de Coq pour ce type de travail. Quelques discussions avec Véronique Benzaken et Évelyne Contejean ont fourni un élan initial dans cette direction.

Ce projet a commencé en 2014 avec Martin Hirzel, Avraham Shinnar et Jérôme Siméon. Le premier problème était de comprendre la sémantique du langage source et de voir comment il pouvait se ramener à des modèles classiques de base de données [28]. Le premier prototype Coq qui a servi de base au projet faisait environ 7k lignes de spécification et 10k lignes de preuves.<sup>1</sup>

Joshua Auerbach et Louis Mandel ont rejoint le projet à partir de 2015. Chacun des membres de l’équipe travaillait parallèlement sur d’autres projets. Stefan Fehrenbach a fait un stage de trois mois

---

1. Le nombre de lignes de code est calculé en utilisant `coqwc` pour Coq et `cloc` pour les autres langages.

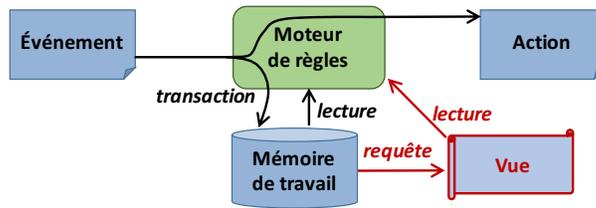


FIGURE 1 – Gestion d’événements avec requêtes dans ODM Insights.

sur le projet mais sur une partie déconnectée du prototype pour le produit. Ainsi, une estimation grossière de l’effectif travaillant à plein temps sur le projet est de 4 années-personnes.

Le code source Coq fait aujourd’hui environ 35k lignes de spécification et 42k lignes de preuve. Le code OCaml extrait fait environ les 80k lignes de code complété par environ 8k de code OCaml pour les différents parsers, printers et autres utilitaires. L’intégration avec ODM Insight est composée de plusieurs parties: (i) environ 1.4k lignes de Java pour traduire la représentation de requêtes internes à ODM Insights vers le langage d’entrée compris par notre compilateur, (ii) environ 400 lignes de Java pour l’intégration du compilateur, (iii) environ 400 lignes de code JavaScript et 550 lignes de Java pour supporter l’exécution des requêtes.

Le compilateur de requêtes d’ODM Insights vers Cloudant a été traduit manuellement en du code Java pour pouvoir être transmis à l’équipe produit. Cette traduction a pris à une personne environ 3 semaines. La taille du compilateur ainsi obtenu est de 11k lignes de code Java (incluant les 1.4k lignes de Java déjà écrites pour la traduction de la représentation interne à ODM Insights). Ce code Java est beaucoup plus petit que la spécification Coq pour principalement deux raisons. D’une part, le compilateur en Java ne contient que les définitions des arbres de syntaxe abstraite des langages intermédiaires et les fonctions de traduction entre ces langages. Ce code ne comprend pas la définition de la sémantique de chacun de ces langages. D’autre part, le prototype Coq nous ayant servi de plate-forme de recherche, il supporte plus de langages d’entrée et de sortie que sa version en Java.

Nous revenons sur les objectifs du projet et présentons l’architecture du compilateur dans la section 2. La section 3 décrit l’utilisation conjointe de preuves et de tests lors du développement du prototype. La section 4 discute des aspects pratiques, ou moins pratiques, de Coq dans un contexte de prototypage industriel. La section 5 présente la méthodologie employée pour le portage du prototype vers Java permettant son intégration dans le produit. Nous concluons le papier avec un état de l’art dans la section 6 et par quelques remarques sur les avantages mais aussi les coûts d’une telle approche dans la section 7.

La version recherche du compilateur en Coq est disponible sous le patronyme Q\*cert à l’adresse suivante : <https://github.com/querycert/qcert>. Au cours de l’article, nous indiquons avec le symbole  un lien vers le code Coq correspondant à la partie du projet décrite.

## 2. Compilateur de requêtes vers Cloudant pour ODM Insights

Avant de parler de la méthodologie utilisée pour développer notre prototype, nous commençons par présenter l’objectif du projet et l’architecture du compilateur.

### 2.1. Requêtes dans ODM Insights

Au cœur d’ODM Insights se trouve un système de gestion de règles métier écrites dans le langage *JRules*. L’exemple suivant illustre la structure événement-condition-action d’une règle métier, qui se

compose d'un évènement dans la clause *when*, d'une condition dans la clause *if* et d'une action dans la clause *then* (omise). Cette règle est exécutée à la réception de la commande d'un client, si l'état du compte de ce client est vide et le coût de cette commande excède la moyenne du coût des commandes.

```
when a purchase occurs, called LAST
if the customer status of 'the customer' is NONE
    and the total amount of all order items
        in the order items of LAST is more than 'average order amount'
then ...
```

La structure des règles facilite l'emploi d'une syntaxe proche de celle d'une langue naturelle, ce qui les rend utiles pour des applications écrites en collaboration entre programmeurs et spécialistes métier. Elles permettent aux entreprises d'identifier de nouvelles opportunités (e.g. vente, marketing), des risques (e.g. détection de fraude, conformité), ou de réagir rapidement à certains événements (e.g. transaction client, notification de livraison).

JRules est une évolution moderne des langages de règles issus des travaux sur les systèmes experts dans les années 70, notamment ceux sur OPS5 [13]. L'architecture du moteur de règles est illustrée dans la figure 1. Le moteur de règles maintient une notion de *mémoire de travail*, contenant un ensemble de *faits* qui peuvent être utilisés par les règles. Lors de la réception d'un événement (e.g., une commande dans la règle précédente), un fait est ajouté dans la mémoire de travail et les règles sont réévaluées en prenant en compte le nouvel état de la mémoire.

Les règles peuvent faire référence à des valeurs calculées globalement sur la mémoire de travail. Dans l'exemple précédent, c'est le cas de 'average order amount'. Cette valeur est spécifiée par la règle suivante qui calcule la moyenne du coût des commandes pour les commandes ayant été accomplies et contenant au moins deux articles.

```
define 'average order amount'
  as the average amount of the order items of all orders,
  where the fulfillment of the order is true
    and the number of elements in the order items of each order is at least 2,
  evaluated every day at 9:28:58 AM
```

Cette règle correspond à une requête sur l'ensemble de la mémoire de travail. Comme le système doit répondre rapidement aux événements, il n'est pas possible d'exécuter les requêtes pour chacun d'entre eux. Celles-ci sont compilées, puis matérialisées sous forme de *vues* qui sont mises à jour, soit périodiquement, soit incrémentalement. La partie en bas à droite de la figure 1 montre la partie du système qui gère les requêtes.

La but de notre prototype est de savoir compiler ce type de requêtes lorsque la mémoire de travail est implantée au dessus de la bases de données Cloudant [9]. Au début du projet, ODM Insights ne supportait l'exécution de requêtes que lorsque la mémoire de travail était une mémoire cache distribuée en Java.

## 2.2. Architecture du compilateur

JRules utilise des objets Java comme modèle de données. Cloudant utilise JSON comme modèle de données et exprime les requêtes sous la forme de tâches map/reduce contenant du code JavaScript. Étant donnée la nature très différente de ces deux langages, le compilateur repose sur de nombreuses représentations intermédiaires. La figure 2 présente le chemin de compilation complet.

**Frontend.** La premier problème est de comprendre le langage source. JRules est un terme générique qui recouvre une famille de langages utilisés dans ODM Insights. Ces différents langages sont tous

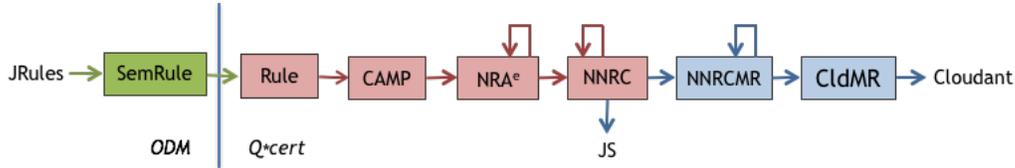


FIGURE 2 – Chemin de compilation de JRules à Cloudant

traduits vers un langage interne commun appelé *SemRule*. Illustrons ce langage sur l'exemple de la requête 'average order amount' en utilisant la syntaxe concrète de SemRule:

```

1  query 'average order amount' {
2    packageExecutionName = ""
3    ruleExecutionName = "average_order_amount"
4    localQueryResult : aggregate {
5      collectclass1 :
6        Order(collectclass1.fullfillment
7          && IlrCollectionUtil.getSize(collectclass1.orderItems) >= 2);
8      local0 : OrderItem() in collectclass1.orderItems;
9    }
10   do {
11     avg {Double.valueOf(((double)local0.amount)).doubleValue()};
12   }
13 }
  
```

Sans nécessairement essayer de comprendre tous les détails de la syntaxe, soulignons quelques aspects importants. La sémantique des règles métier, héritée de la programmation logique, se fonde sur une notion d'unification entre les expressions du langage et les objets dans la mémoire de travail. Dans l'exemple, les lignes 5 à 7 utilisent la notation '*var* : *ClassName*(*cond*)' qui signifie: si un objet dans la mémoire de travail est sous-type de la classe *ClassName* et vérifie la condition *cond*, alors il faut instancier la variable *var* avec l'objet correspondant. Comme on peut l'observer dans l'expression de la ligne 8, l'unification peut utiliser une variable définie auparavant dans la règle (*collectclass1*) et accéder aux champs d'un des objets (*orderItems*).

Les requêtes sont écrites comme les autres règles, mais utilisent une extension qui permet d'appliquer cette logique d'unification sur *l'ensemble des faits* en mémoire, plutôt que sur chaque fait individuellement. Dans l'exemple, il s'agit de l'expression `aggregate { ... } do { ... }` qui va de la ligne 4 à la ligne 12. La clause `aggregate` accumule toutes les combinaisons d'objets qui unifient avec les conditions, et retourne un ensemble de variablesinstanciées. La clause `do` contient une expression qui peut accéder à ces variables, et une fonction d'agrégation, dans l'exemple `avg`. L'expression `aggregate` est une forme de compréhension [30, 29].

Afin de capturer la sémantique de JRules en Coq, nous avons défini les langages Rules  $\clubsuit$  et son noyau CAMP  $\clubsuit$  (*Calculus for Aggregating Matching Patterns*) [28]. CAMP repose sur du filtrage par motif. Il permet notamment de représenter les notions d'unification sur les objets Java et les expressions d'agrégation décrites précédemment.

**Middle-end.** Pour le cœur du compilateur, nous avons décidé de nous reposer sur des représentations bien connues en bases de données. En effet, développer un optimiseur de requêtes est une tâche non triviale, que ce soit en terme de temps de développement, d'architecture, mais également en terme de définition de l'ensemble des optimisations nécessaires. Dans ce contexte, la possibilité de

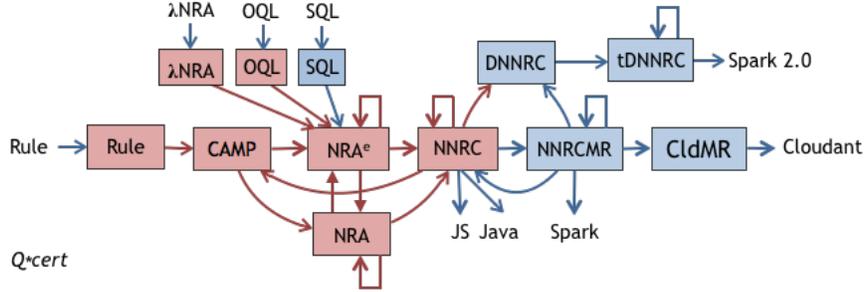


FIGURE 3 – Le compilateur complet en Coq

se fonder sur la littérature issue des travaux de bases de données (voir par exemple [15, 31, 25]) est un critère important qui a guidé le choix des représentations intermédiaires.

Nous avons donc choisi de traduire CAMP vers NRA  $\clubsuit$  (*Nested Relational Algebra* ou NRA), l’algèbre relationnelle imbriquée [10]. Cette représentation à base de combinateurs élimine les variables et simplifie les optimisations. Néanmoins, les requêtes venant de CAMP se sont révélées difficiles à optimiser. Nous avons donc introduit NRA<sup>e</sup>  $\clubsuit$ , une extension de l’algèbre relationnelle imbriquée qui aide à la gestion de l’environnement et simplifie donc les optimisations [2].

**Backend.** Afin de pouvoir produire simplement du code impératif, nous traduisons NRA<sup>e</sup> vers NNRC  $\clubsuit$  (Named Nested-Relational Calculus) [32]. NNRC correspond à un petit langage fonctionnel (d’ordre zéro) avec des listes en compréhension pour lequel il est aisé de produire du code JavaScript  $\clubsuit$ .

NNRC nous sert également de base pour l’introduction du modèle *map-reduce*. NNRC Map-Reduce  $\clubsuit$  (NNRCMR) est un langage où tous les calculs sont exprimés sous forme de chaînes de fonctions *map* et *reduce* définies elles même en NNRC. À partir de ce niveau dans le compilateur, nous avons donc introduit une notion d’évaluation distribuée.

Le dernier langage intermédiaire est CldMR  $\clubsuit$ . Cela reste un modèle *map-reduce* où le corps des fonctions est écrit en NNRC, mais cette fois-ci, le modèle prend en compte les spécificités de Cloudant. Par exemple, les fonctions *reduce* dans Cloudant ont une forme plus contrainte : il y a trois fonctions de réductions prédéfinies (**sum**, **count**, **stats**) et sinon, les fonctions doivent pouvoir être appelées un nombre arbitraire de fois afin de construire incrementalement le résultat.

De CldMR, la traduction vers Cloudant est directe. La structure de map-reduce est celle attendue pour Cloudant. Pour le code des fonctions, nous réutilisons le compilateur de NNRC vers JavaScript.

**Support d’autres langages.** Nous avons également utilisé ce développement pour expérimenter avec d’autres langages à des fins purement de recherche. Le compilateur supporte notamment en entrée un sous-ensemble de SQL et OQL [4], et peut émettre du code pour Spark [33]. L’ensemble des langages et les chemins de compilation sont présentés dans la figure 3.

### 3. Preuves et Tests

Le résultat de correction souhaité pour le compilateur est que pour chaque requête SemRule pour laquelle du code Cloudant est produit, le résultat de l’évaluation de la requête sur Cloudant est le même que celui produit par l’implantation courante d’ODM Insights.

Par construction, cette propriété ne peut pas être prouvée formellement en Coq puisque la sémantique d'ODM Insights est donnée par son implantation en Java et celle de Cloudant par son implantation en Erlang. Nous nous reposons donc sur un mélange de preuves et de tests pour nous convaincre de la correction de notre compilateur.

Dans la suite de la section, nous décrivons pour chaque partie du compilateur les techniques qui ont été utilisées pour la valider. Les couleurs sur la figure 2 distinguent les parties prouvées (en rouge), des parties non prouvées (en bleu).

**Langages Intermédiaires.** Illustrons tout d'abord comment les langages sont définis. Pour cela, prenons l'exemple de NRA<sup>e</sup>. Premièrement, nous définissons l'arbre de syntaxe abstraite du langage en utilisant un type inductif :<sup>2</sup> ❀

```
Inductive algenv : Set :=
| ANID : algenv
| ANConst : data -> algenv
| ANBinop : binOp -> algenv -> algenv -> algenv
| ANUnop : unaryOp -> algenv -> algenv
| ANMap : algenv -> algenv -> algenv
...

```

Ensuite, la sémantique est définie comme une fonction d'évaluation : ❀

```
Fixpoint fun_of_algenv (op:algenv) (env: data) (x:data) : option data :=
match op with
| ANID => Some x
| ANConst rd => Some (normalize_data h rd)
| ANBinop bop op1 op2 =>
  olift2 (fun d1 d2 => fun_of_binop h bop d1 d2)
        (fun_of_algenv op1 env x) (fun_of_algenv op2 env x)
| ANUnop uop op1 =>
  olift (fun d1 => fun_of_unaryop h uop d1) (fun_of_algenv op1 env x)
| ANMap op1 op2 =>
  let aux_map d :=
    lift_oncoll (fun c1 => lift dcoll (rmap (fun_of_algenv op1 env) c1)) d
  in olift aux_map (fun_of_algenv op2 env x)
...
end.

```

Cette fonction est paramétrée explicitement par la requête `op`, l'environnement `env` et une donnée d'entrée `x`, et implicitement par la hiérarchie de type `h` et par un environnement constant `c` correspondant à la mémoire de travail.

Les fonctions d'évaluation permettent de faire du test et servent également de spécification pour les preuves.

**Frontend.** SemRule n'ayant pas de sémantique formelle, le frontend est vérifié à l'aide de tests. Le résultat de l'évaluation obtenue par l'interpréteur de Rule est comparé à celui obtenu par l'évaluation à l'intérieur d'ODM Insights. La base de tests utilisée est celle d'ODM Insights.

L'une des fonctions importantes de l'infrastructure de tests concerne la manipulation des données d'entrées et du résultat attendu. Les données d'entrées et le résultat sont importés dans le code d'ODM, notamment pour être typées, puis exportées dans un format JSON qui inclut des annotations de type. Ce format JSON est ensuite utilisé par l'interpréteur de Rule et pour comparer les deux résultats.

2. Pour des raisons historiques, l'AST de NRA<sup>e</sup> s'appelle `algenv`.

La traduction de Rule vers CAMP a été prouvée formellement. Ainsi, afin de déboguer la traduction initiale de SemRules, nous avons pu aussi réaliser les tests au niveau du noyau CAMP.

Nous avons développé une version instrumentée de l'interprète de CAMP qui en plus de l'évaluation du programme, conserve de l'information sur l'environnement d'exécution. Quand une erreur est détectée, cet interprète retourne une version annotée du programme d'entrée qui identifie précisément la position et la raison du problème (principalement des erreurs de type). Cette version de l'interprète est prouvée équivalente à la version d'origine (qui définit la sémantique) modulo les informations supplémentaires en cas d'erreur. Cet interprète s'est révélé très utile lors de la mise au point du frontend lorsqu'il produisait du code CAMP incorrect.

**Middle-end.** Le cœur du compilateur est entièrement en Coq et donc n'a pas les problèmes de frontière du frontend. Cette partie qui va de CAMP à NNRC optimisé a été entièrement prouvée.

Pour illustrer ce qui nous a motivé à mettre l'effort nécessaire à la réalisation des preuves du middle-end, considérons deux équivalences classiques de l'algèbre relationnelle [31]. Ces équivalences sont utilisées dans l'optimiseur de  $NRA^e$  ( $\sigma$  et  $\cup$  correspondent à la sélection et à l'union ensembliste) :

$$\begin{aligned}\sigma_{p_1 \wedge p_2}(q) &\equiv \sigma_{p_1}(\sigma_{p_2}(q)) \\ \sigma_{p_1 \vee p_2}(q) &\equiv \sigma_{p_1}(q) \cup \sigma_{p_2}(q)\end{aligned}$$

En dépit de leur simplicité, aucune de ces deux équivalences n'est vraie! Plus précisément, elles ne sont vraies que sous certaines hypothèses qui doivent être garanties par l'optimiseur. La première équivalence n'est vraie que pour une requête bien typée, c'est-à-dire en l'absence d'erreur à l'exécution. La seconde est vraie pour l'union ensembliste, mais dans le cas des multi-ensembles<sup>3</sup> elle n'est vraie que pour l'union dite maximale [20]. Prenons par exemple l'application de la première équivalence relationnelle dans un exemple mal typé (ici *age* est un nombre plutôt qu'une chaîne de caractères) :

$$\sigma_{age="John" \wedge false}(q) \not\equiv \sigma_{age="John"}(\sigma_{false}(q))$$

L'évaluation de l'expression de gauche échoue, mais celle de droite retourne un ensemble vide. Il s'agit d'un exemple très simple, mais qui montre qu'il est relativement facile d'oublier les préconditions fondamentales lors de l'implantation d'une règle de réécriture. On peut noter que le chapitre correspondant dans [31] ne mentionne pas les aspects liés au typage.

L'optimiseur de  $NRA^e$  est composé de plus de soixante-dix de ce type de réécritures. Les preuves ne sont généralement pas difficiles. Elles permettent de mettre en avant des préconditions subtiles et également de capturer les erreurs d'inattention qui arrivent facilement dans ce type de code. Donc bien que la preuve de correction de l'optimiseur ne soit pas nécessaire, pouvoir reposer sur la garantie de l'absence de bogue dans cette partie a été d'un grand bénéfice.

Pour prouver la correction des traductions entre deux langages, nous utilisons les fonctions d'évaluation qui définissent leurs sémantiques. Illustrons cela avec la traduction de CAMP vers  $NRA^e$ . La sémantique de CAMP est définie par la fonction `interp`  et la sémantique de  $NRA^e$  est définie par la fonction `fun_of_algenv` . La fonction de traduction en CAMP à  $NRA^e$  est `algenv_of_pat` . À partir de ces trois fonctions, on peut énoncer la propriété de correction de la traduction: 

**Lemma** `pat_envtrans_correct` `h c q env d` :

```
lift_failure (interp h c q env d) = fun_of_algenv h c (algenv_of_pat q) (drec env) d.
```

Ce lemme est paramétré par la hiérarchie de type `h`, un environnement constant `c` correspondant à la mémoire de travail, la requêtes d'entrée `q`, l'environnement d'exécution `env` et une donnée d'entrée `d`. Il illustre qu'il y a un encodage différent de l'environnement dans les deux sémantiques : `env` (une

3. La sémantique de SQL et celle de la plupart des langages de requêtes utilisés en pratique, y compris dans JRules, se fonde sur une notion de multi-ensemble.

liste associative) devient `drec env` (un enregistrement). De même, pour le résultat de l'évaluation, les notions d'erreur dans CAMP et dans  $\text{NRA}^e$  étant encodées différemment, la fonction `lift_failure` permet de faire le lien entre ces deux encodages.

Même si cette partie est entièrement prouvée, il reste bien sûr possible que le code ne fasse pas ce qu'il est supposé faire à cause de spécifications erronées ou incomplètes. Comme les spécifications sont données par les interprètes, il a été important de les déboguer. La réalisation de preuves est pour cela un outil très efficace et qui force à regarder les spécifications en détail. Par ailleurs, comme on a prouvé la correction de la traduction avec CAMP et que la sémantique de CAMP a été testée, cela veut dire que les sémantiques des langages suivants sont aussi corrects vis-à-vis des tests.

D'un point de vue de la complexité du développement en Coq, plusieurs points sont à relever. Les preuves ayant demandé le plus de travail sont sans aucun doute celles dédiées à deux aspects sous-jacents de l'infrastructure: le système de type  $\clubsuit$  et les opérations sur le renommage et la substitution pour NNRC  $\clubsuit$ . Le système de type est lui-même relativement complexe, incluant notamment deux notions d'enregistrements (ouvert/fermé), et une hiérarchie de classes. Il est également essentiel au reste du compilateur car l'absence d'erreurs de typage est une précondition dans la plupart des réécritures pour  $\text{NRA}^e$  et pour NNRC. Dans le reste du code, la difficulté a plus souvent été de choisir la bonne formulation des propriétés à prouver que les preuves proprement dites. Un exemple important dans cette catégorie est celle de la définition de la notion d'équivalence entre deux expressions pour  $\text{NRA}^e$   $\clubsuit$  ou NNRC  $\clubsuit$ .

**Backend.** La correction des traductions de NNRC à CldMR est vérifiée par un mélange de tests et de preuves. À terme, nous espérons prouver complètement ces traductions. Pour le moment, quelques propriétés importantes de la traduction de NNRC à NNRCMR ont été prouvées et les optimisations sur NNRCMR sont prouvées correctes individuellement, mais leur composition n'a pas été prouvée correcte.

La correction des générateurs de code vers JavaScript et Cloudant repose sur du test. Leur mise au point a été faite principalement en utilisant les outils et techniques classiques pour les langages cibles. Par exemple, en JavaScript il est facile d'ajouter des *printf*. Le débogage du code Cloudant, un système distribué exécuté sur le Cloud, s'est s'avéré une tâche beaucoup plus difficile. Pour répondre à ce problème, nous avons réalisé un backend naïf de NNRCMR vers Spark [33] (où les fonctions `map` et `reduce` sont en JavaScript) pour aider à la mise au point du code.

Il est à noter que même si dans cette partie nous avons utilisé Coq principalement comme un simple langage fonctionnel, les aspects de preuves ont quand même été utiles. Des propriétés ont été prouvées seulement comme un moyen très efficace de recherche de bogues. Mais parfois la présence de preuves nous a gêné. En effet, cette partie du compilateur était moins mature que le reste du projet et a beaucoup bougé. Maintenir les preuves pouvait être fastidieux et donc parfois elles ont été simplement supprimées.

## 4. Coq en pratique

Dans un contexte de prototypage industriel, l'utilisation de Coq change l'expérience de développement de façon considérable. Nous revenons ici sur les aspects ayant demandé le plus d'attention de la part de l'équipe.

### 4.1. Communication avec Java

Dans la figure 2, la partie à gauche de la barre verticale est écrite en Java et fait partie d'ODM Insights. La partie à droite de la barre est écrite en Coq et fait partie de notre prototype de recherche.

La flèche traversant la frontière fait également partie de notre code mais est écrite en Java.

Nous avons utilisé trois techniques différentes pour communiquer entre Java et Coq. La première est par la génération de fichiers sources Coq. Le traducteur de SemRule vers CAMP émet du code Coq qui contient la requête directement sous forme d'une valeur Coq correspondant à l'arbre de syntaxe. Cette technique a été particulièrement utile au début du processus de développement. Elle permet d'évaluer des exemples dans le mode interactif de Coq et ainsi de faire du test unitaire de fonctions.

La seconde technique de communication entre Java et Coq est par un fichier texte. Nous avons défini une syntaxe concrète pour CAMP qui sert de format d'échange. Pour utiliser cette technique, le compilateur Coq est extrait en OCaml et est combiné avec un parseur écrit avec Menhir [27]. Cette technique produit un compilateur en ligne de commande qui prend en entrée des requêtes écrites en CAMP, nous donnant un moyen simple d'écrire des requêtes sans passer par ODM Insights.

Enfin, la troisième méthode de communication entre Java et Coq est d'utiliser OCamlJava [8], un compilateur d'OCaml vers du bytecode Java. À partir du code Coq extrait en OCaml, nous avons défini une bibliothèque qui peut être compilée avec OCamlJava et permet ainsi d'appeler le compilateur directement depuis Java. Nous reviendrons sur l'utilisation d'OCamlJava dans la section 5.

## 4.2. Types et fonctions externes

Le modèle de données implémenté en Coq inclut des types primitifs (entiers naturels, booléens et chaînes de caractères). Ces types de données ne sont pas suffisants pour compiler les requêtes issues du produit. Il est donc important de pouvoir ajouter de nouveaux types primitifs et des opérations sur ces types. Dans le cas d'ODM, il s'agit principalement d'opérations sur les flottants et sur les dates.

Pour résoudre ce problème, nous avons rendu notre compilateur extensible. Le but est de rendre possible l'ajout de nouveaux types primitifs sans obliger à systématiquement réviser l'ensemble des preuves. Pour cela nous utilisons les classes de type. En pratique, cela veut dire que le code du compilateur est paramétré par de multiples classes de type qui décrivent les points d'extension pour les types externes et les opérations sur ces types.

Cette solution résout le problème, mais rajoute de la complexité à l'ensemble du code. Pour simplifier l'extraction (comme le code extrait devrait fournir explicitement les instances des classes de type à chaque appel de fonction), nous avons créé des foncteurs qui en interneinstancient les classes de type. Cette partie du développement a parfois été une source de frustration pour l'équipe. Elle ajoute un niveau de complexité inattendue lors du développement. Les messages d'erreur de typage lors de l'utilisation des classes de types peuvent être difficiles à comprendre. La navigation dans le code est aussi rendu plus difficile.

## 4.3. Notations

Plusieurs de nos représentations intermédiaires (comme NRA ou NNRC) sont issues de langages présents dans la littérature. Comme Coq permet de définir des notations qui incluent des symboles unicode, il a semblé naturel de réutiliser autant que possible les notations classiques, comme par exemple  $\sigma$  pour l'opération de sélection et  $\cup$  pour l'union. Comme certaines de ces notations sont disponibles dans plusieurs langages de notre compilateur, cela a également conduit à utiliser la notion de portée des notations de Coq pour distinguer quels sont les langages en cours d'utilisation.

L'utilisation de ces notations a conduit à des désaccords à l'intérieur de l'équipe! Bien qu'il soit très agréable que les théorèmes en Coq ressemblent à leurs formulations dans les articles, il est parfois nécessaire de se référer à la version ASCII des noms à l'intérieur des preuves. Cela oblige donc à manipuler un double ensemble de noms. Par ailleurs, certaines des notations étaient évidentes pour certains des membres de l'équipe et obscures pour d'autres. En fait, tous les auteurs ont probablement eu chacun de ces sentiments sur des notations différentes.

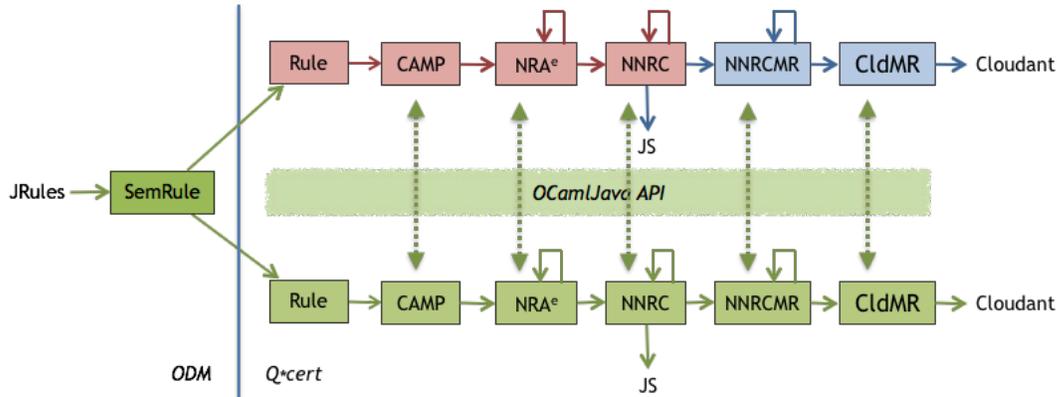


FIGURE 4 – Double chaîne de compilation en OCaml et Java.

Il y a aussi eu un usage intensif (parfois inconscient) de symboles unicode dans des noms d'identificateurs. Par exemple, si une fonction prend deux types en entrée, nous pouvons les appeler  $\tau_{in_1}$  and  $\tau_{in_2}$  (en utilisant les nombres unicode en indice). Cela rend l'écriture du code plus lourde. De plus, lors de l'extraction vers OCaml, les noms sont modifiés pour produire des identificateurs valides en OCaml. Enfin, cela rend le copier/coller de code entre article et code source (dans les deux sens) plus difficile.

## 5. Ré-implémenter le compilateur en Java

Une fois le compilateur en Coq développé, la question devient comment produire une version de ce compilateur prête à être intégrée au produit. En pratique, tout logiciel est maintenu par une équipe qui partage un savoir commun sur son implantation et sur les outils utilisés pour le développer.

Dans le cas d'ODM Insight, la quasi-totalité du code en production est écrite en Java, nous avons donc décidé de traduire manuellement le code Coq vers du code Java idiomatique. Cette traduction pose deux problèmes : (1) les preuves de correction faites en Coq ne sont pas préservées par la traduction et de nouvelles erreurs peuvent facilement être introduites ; (2) une fois le code Java dans les mains de l'équipe de développement, et le compilateur Coq dans l'équipe de recherche, les deux versions peuvent diverger. Aucun de ces problèmes n'a de solution idéale, mais nous avons mis en place des techniques de développement pour palier à ces problèmes.

Pour palier à l'impossibilité de conserver directement les preuves, le code Java conserve la même structure que le code Coq. De plus, nous vérifions empiriquement que l'implantation Coq et Java de chaque composant produit les mêmes réponses pour des entrées identiques sur une base de tests conséquente. L'évolution indépendante du code Coq et du code Java étant inévitable, nous avons structuré le code Java de telle sorte qu'il soit facile de le comparer manuellement avec le code Coq original. Cela rend possible d'identifier puis de porter les changements d'une version à l'autre, même si cette tâche n'est pas automatique.

Un autre but important est celui d'écrire du Java « naturel » afin qu'il puisse être facilement compris et adopté par l'équipe produit. Cet objectif peut parfois rentrer en conflit avec celui d'écrire du Java qui puisse être comparé avec le code en Coq.

**Assurer une traduction exacte** Afin de mener à bien la traduction du code Coq en Java, nous avons mis en place une infrastructure qui permet de traduire le code progressivement et de tester la correction de la traduction. Cette infrastructure est présentée dans la figure 4.

Le chemin de compilation en haut de la figure 4, reprend celui de la figure 2. Il s’agit de l’implantation d’origine en Coq. OCamlJava a été utilisé pour tester ce chemin de compilation à partir de Java, et l’intégration avec le code d’ODM Insight. Grâce à cette intégration, nous avons également pu utiliser la base de tests complète d’ODM Insight. À partir de là, nous avons une implantation de référence en place et nous avons pu commencer la traduction manuelle du compilateur vers Java.

Le chemin de compilation du bas de la figure 4 représente la ré-implantation manuelle du compilateur en Java. Les arbres de syntaxe abstraite de chacun des langages sont implémentés par des hiérarchies de classes Java au lieu des types algébriques de Coq.

Les lignes verticales pointillées représentent la possibilité à chaque étape de compilation d’une requête de passer d’une représentation Coq à une représentation Java. La représentation interne de chaque arbre de syntaxe abstraite est différente mais isomorphe et partage la même sémantique. La communication entre les deux représentations se fait en passant par une représentation des arbres sous forme de S-expression. Il est ainsi possible de prendre des chemins arbitraires dans cette double chaîne de compilation.

Il y a deux avantages clés à cette double chaîne de compilation. Premièrement, chaque étape en Java peut être créée et déboguée indépendamment. Deuxièmement, une étape donnée peut être exécutée à la fois en Java et en Coq et les structures des arbres de syntaxe abstraite (ASTs) produits peuvent être comparées. Lors d’un test de bout en bout, quand une seule étape est en Java et les autres restent en Coq, il est facile d’attribuer l’erreur au code en Java. Si plusieurs étapes avaient été testées simultanément, il aurait été plus difficile d’identifier l’erreur. De plus, plusieurs erreurs auraient pu se compenser et ainsi cacher des erreurs de traduction.

Par ailleurs, la comparaison des ASTs après chaque étape fournit une confiance forte dans la fidélité du comportement du code Java par rapport au code Coq. Elle permet de trouver des erreurs qui produisent du code correct mais pas forcément identique (par exemple des optimisations manquantes). Une comparaison textuelle des ASTs sous forme de S-expression aurait été trop restrictive à cause du choix des noms de variables qui peut être différent selon le chemin de compilation.

La réutilisation des tests d’ODM Insight avec notre double chaîne de compilation nous donne donc une grande confiance dans l’exactitude de notre traduction de Coq vers Java. On peut noter qu’une approche très différente serait de développer une forme alternative d’extraction pour Coq directement vers Java. Bien que le code source de l’extraction soit relativement petit et bien isolé, cette idée aurait probablement été une distraction trop importante en cours de projet, mais reste une possibilité intrigante pour l’avenir.

**Assurer une optimisation exacte.** Quand nous avons essayé de déboguer la version Java de notre compilateur, nous nous sommes rapidement rendu compte que nous n’avions pas une bonne intuition de quelles optimisations étaient appliquées et dans quelle ordre. Pour résoudre ce problème, nous avons instrumenté les optimiseurs pour tracer ces informations.

Comme cela était simplement pour des raisons de débogage, nous n’avons pas fait cela de la « bonne » façon.<sup>4</sup> À la place, nous avons ajouté dans le code un appel d’une fonction spécifiée par une classe de type. Normalement, cette classe est instanciée par une fonction sans effet. Mais quand nous voulons tracer les optimisations qui sont appelées (pour comparaison avec la version Java), nous instancions la classe avec une fonction qui, lors de l’extraction vers OCaml, trace les informations sur la sortie standard. Bien que théoriquement l’ajout d’effets de bord casse la correction du système Coq, ici cela n’est fait qu’en mode débogage (et en pratique ne pose pas de problèmes).

---

4. En utilisant une monade d’état qui conserve la liste des optimisations appliquées.

Cette trace peut être comparée avec la trace équivalente obtenue lors de l'optimisation en Java pour confirmer que les mêmes optimisations sont utilisées dans le même ordre.

**Garder le lien vers Coq en écrivant du code Java idiomatique** La ré-implantation du compilateur a trois aspects : (1) la représentation des ASTs en Java, (2) la réécriture des fonctions de traduction, et (3) la réécriture des optimisations.

En Coq, les ASTs sont représentés par des types algébriques (voir section 2). En Java, il n'y a pas de types algébriques et donc la programmation orientée objet doit être utilisée. Par exemple, pour le langage NRA<sup>e</sup>, Java définit la class abstraite `NraNode` avec les sous-classes `NraProduct`, `NraUnaryOperator`, etc. Chaque sous-classe a son propre constructeur, ses variables d'instance et ses accesseurs.

Nous illustrons maintenant l'usage de ces structures de données sur un exemple. Nous avons choisi la réécriture présentée dans la section 3.

```
/**
 * From TOptimEnvFunc.v: last checked 5/2/2016
 * Definition tselect_and_fun {fruntime:foreign_runtime} (p: algenv) :=
 *   match p with
 *     ANSelect op1 (ANSelect op2 op) =>
 *     ANSelect (ANBinop AAnd op2 op1) op
 *   | _ => p
 *   end.
 */
private static class tselect_and_fun implements OptFun {
  @Override
  public NraNode optimize(NraNode nra) {
    if (nra instanceof NraSelect) {
      NraNode op1 = nra.getOperand1();
      NraNode select = nra.getOperand2();
      if (select instanceof NraSelect) {
        NraNode op2 = select.getOperand1();
        NraNode op = select.getOperand2();
        return new NraSelect(new NraBinaryOperator(BinaryOperator.And, op2, op1), op);
      }
    }
    return nra;
  }
}
```

Chaque fonction traduite en Java contient le code Coq correspondant dans l'entête du commentaire (les autres commentaires, dédiés aux programmeurs Java, sont omis). Si la fonction Coq est longue, l'entête peut être raccourci et le reste du code Coq est alors mis en commentaire dans le corps de la fonction à côté du code Java correspondant. Cela permet aux programmeurs comprenant Coq de voir l'intention de la traduction et aussi de vérifier si une mise à jour est nécessaire. L'information `last checked` dans l'entête permet de retrouver dans le dépôt du code Coq les changements réalisés depuis la dernière synchronisation entre le code Java et le code Coq. Dans le code Coq, le lien vers le code Java est également indiqué.

L'exemple montre aussi comment nous traduisons le filtrage de motif en Java. Ici, nous utilisons simplement la directive `instanceof` de Java. Mais dans des cas plus complexes, nous utilisons des fonctions utilitaires. Considérons le fragment de code Coq suivant :

```
match p with
  ANProduct (ANUnop AColl (ANUnop (ARec s1) p1))
    (ANUnop AColl (ANUnop (ARec s2) p2)) => ...
```

Notre traduction en Java est la suivante :

```

if (nra instanceof NRAProduct) {
    NRANode unop1 = nra.getOperand1();
    NRANode unop2 = nra.getOperand2();
    if (hasKind(unop1, AColl) && hasKind(unop2, AColl)) {
        NRANode sub1 = unop1.getOperand(); // (ANUnop (ARec s1) p1)
        NRANode sub2 = unop2.getOperand(); // (ANUnop (ARec s2) p2)
        if (hasKind(sub1, ARec) && hasKind(sub2, ARec)) ...
    }
}

```

La fonction `hasKind` vérifie que son premier argument est un `NRAUnaryOperator` et ensuite vérifie qu'il correspond à l'opérateur donné en deuxième argument. Nous utilisons les énumérations de Java pour donner des noms lisibles aux opérateurs (comme `BinaryOperator.And`) et nous définissons également quelques alias comme :

```
private static final UnaryOperatorKind AColl = UnaryOperatorKind.SingletonBag;
```

Ces noms courts sont indispensables pour rendre le code du filtrage de motif clair. Le choix des fonctions utilitaires s'est fait par essai-erreur. Les premières versions des fonctions ré-implantées étaient beaucoup plus verbeuses et difficiles à lire jusqu'à ce que nous ayons défini une bibliothèque de fonctions utilitaires raisonnables.

D'autres défis (pas directement montrés par cet exemple) ont dû être résolus lors de la ré-implantation. Par exemple, Java est un langage impératif pour lequel les fonctions de bibliothèques sont rarement purement fonctionnelles. Par exemple, le tri d'un tableau se fait en place ou la fusion de deux dictionnaires peut modifier l'un des deux. Pour toutes ces fonctions, nous avons ré-implanté une version qui introduit une copie afin de leurs donner une interface persistante. Il est à noter que la nature impérative de Java a aussi de bons côtés. Par exemple, dans le code Coq, la génération de noms frais requiert de l'effort alors qu'en Java ce n'est pas un problème.

Le projet étant jeune, il est difficile de savoir si les techniques mises en place pour éviter la divergence de code vont être efficaces. Mais nous pensons au moins avoir réussi à obtenir du code Java raisonnable.

## 6. État de l'art

Les progrès considérables réalisés par les assistants de preuves, tels que Coq ou Isabelle, au cours des dix dernières années sont bien connus. Plusieurs projets démontrent qu'il est possible d'utiliser ces outils pour le développement de gros systèmes, que le but soit pour un traitement formel [14] ou pour la certification logicielle [19, 17, 12]. Notre travail se place dans un contexte plus proche du génie logiciel, utilisant Coq comme un outil parmi d'autres pour le développement et la vérification, dans la lignée de [16]. Nous avons trouvé peu de projets similaires dans la littérature existante, ce qui laisse à penser qu'il s'agit encore d'une utilisation relativement marginale dans l'industrie.

Le fait d'avoir choisi Coq pour développer le prototype du compilateur nous a obligé à traduire le code en Java. Nous avons vu que cette traduction et le maintien des deux compilateurs ont un coût non négligeable. Une alternative qu'il aurait été intéressant d'explorer est l'utilisation d'outils qui permettent d'exprimer des propriétés directement sur les programmes Java et de les prouver [5, 23].

Le domaine des bases de données en général, et celui des langages de requêtes en particulier, se caractérisent par des liens forts entre théorie et pratique. Depuis les travaux de Trinder et Wadler [30] dans les années 80, la formalisation des langages de requêtes se repose souvent sur des bases fonctionnelles [29, 32, 11], ce qui les rend particulièrement adaptées à une mécanisation en Coq. Un certain nombre de travaux récents traitent de différents aspects de la formalisation des

systèmes de bases de données (e.g., pour le relationnel [3, 21], XML [6], ou Linq [22]). Ces travaux sont techniquement complémentaires au développement présenté dans cet article.

Le développement d’outils et de méthodes formelles pour la spécification et l’implantation de compilateurs de requêtes est un sujet important dans le contexte relationnel [26]. La complexité liée au développement de ces compilateurs et leur utilisation dans de nouveaux contextes applicatifs, fait que le sujet reste d’actualité [18]. Notre approche est fondamentalement similaire à celle de Coko-Kola [7], un optimiseur de requêtes développé dans les années 90 avec l’aide du prouveur Larch du MIT.

## 7. Conclusion

L’utilisation de Coq pour le prototypage a été intéressante à plusieurs niveaux. Les divers degrés d’expérience avec Coq dans l’équipe au début du projet ajoutaient un nombre d’inconnues en terme de faisabilité et de temps de développement. La décision de ne pas se limiter à une utilisation de Coq comme outil de spécification mais également de développer un prototype complet du compilateur a créé un ensemble de défis supplémentaires. Nous espérons que cet article fournit un aperçu de cette expérience, de ces défis, et que certaines des techniques décrites pourront être utiles en dehors de l’usage spécifique que nous avons pu en faire.

A posteriori, déterminer si l’effort de développement supplémentaire vaut la peine pour un projet unique n’est pas évident. L’utilisation d’un outil comme Coq pour le développement requiert un investissement important, notamment au départ du projet, et représente un coût supplémentaire non négligeable. En contrepartie, reposer sur un noyau entièrement prouvé change l’expérience de développement de façon parfois très positive. Lorsqu’un bogue est identifié, la recherche de son origine peut immédiatement exclure une partie importante du code. Coq incite également à écrire du code réutilisable. Dans notre cas, le compilateur est construit autour de petits langages bien définis. Cet investissement a notamment porté ses fruits pendant l’ajout de langages de requêtes additionnels (OQL et SQL notamment). En nous basant sur cette expérience, nous estimons que la réalisation d’un compilateur de requêtes entièrement certifié est tout à fait réalisable.

**Remerciements.** Nous tenons à remercier chaleureusement Philippe Suter qui nous a aidé à améliorer grandement la qualité de l’article même pendant le weekend de Thanksgiving.

## Références

- [1] M. ARNOLD, D. GROVE, B. HERTA, M. HIND, M. HIRZEL, A. IYENGAR, L. MANDEL, V. SARASWAT, A. SHINNAR, J. SIMÉON, M. TAKEUCHI, O. TARDIEU et W. ZHANG : META: Middleware for events, transactions, and analytics. *IBM R<sup>2</sup>D*, 60(2–3):15:1–15:10, 2016.
- [2] J. AUERBACH, M. HIRZEL, L. MANDEL, A. SHINNAR et J. SIMÉON : Handling environments in a verified query compiler. *In SIGMOD*, 2017. To appear.
- [3] V. BENZAKEN, E. CONTEJEAN et S. DUMBRAVA : A Coq formalization of the relational data model. *In ESOP*, 2014.
- [4] M. BERLER, R. CATTELL et D. BARRY : *The object data standard: ODMG 3.0*. Morgan Kaufman, 2000.
- [5] L. BURDY, Y CHEON, D. COK, M. ERNST, J. KINIRY, G. LEAVENS, R. LEINO et E. POLL : An overview of JML tools and applications. *STTT*, 7(3), 2005.
- [6] J. CHENEY et Ch. URBAN : Mechanizing the metatheory of mini-XQuery. *In CPP*, 2011.
- [7] M. CHERNIACK et S. ZDONIK : Rule languages and internal algebras for rule-based optimizers. *In SIGMOD*, 1996. <http://www.cs.brandeis.edu/~cokokola>.

- [8] X. CLERC : Ocaml-java. <http://ocamljava.org>.
- [9] Anatomy of the Cloudant DBaaS, 2015. <https://cloudant.com/CloudantTechnicalOverview.pdf>.
- [10] S. CLUET et G. MOERKOTTE : Nested queries in object bases. *In DBPL*, 1993.
- [11] L. FEGARAS et D. MAIER : Optimizing object queries using an effective calculus. *TODS*, 25(4), 2000.
- [12] K. FISHER : Using formal methods to enable more secure vehicles: Darpa’s hacms program. *ACM SIGPLAN Notices*, 49(9):1–1, 2014.
- [13] Ch. FORGY : OPS5 user’s manual. Rapport technique 2397, CMU, 1981.
- [14] G. GONTHIER, A. ASPERTI, J. AVIGAD, Y. BERTOT, C. COHEN, F. GARILLOT, S. LE ROUX, A. MAHBOUBI, R. O’CONNOR, S. O. BIHA *et al.* : A machine-checked proof of the odd order theorem. *In ITP*, 2013.
- [15] G. GRAEFE : Query evaluation techniques for large databases. *CSUR*, 25(2), 1993.
- [16] J. S. JORGE, V. M. GULÍAS et D. CABRERO : Certifying properties of programs using theorem provers. *In Verification, Validation and Testing in Software Engineering*, chapitre 10, pages 220–267. IGI Global, 2007.
- [17] G. KLEIN, K. ELPHINSTONE, G. HEISER, J. ANDRONICK, D. COCK, Ph. DERRIN, D. ELKADUWE, K. ENGELHARDT, R. KOLANSKI, M. NORRISH *et al.* : sel4: Formal verification of an os kernel. *In SOSP*, 2009.
- [18] Y. KLONATOS, C. KOCH, T. ROMPF et H. CHAFI : Building efficient query engines in a high-level language. pages 853–864, 2014.
- [19] X. LEROY : Mechanized semantics for compiler verification. *In CPP*, 2012.
- [20] L. LIBKIN et L. WONG : Some properties of query languages for bags. *In DBPL*. 1994.
- [21] G. MALECHA, G. MORRISSETT, A. SHINNAR et R. WISNESKY : Toward a verified relational database management system. *In POPL*, 2010.
- [22] G. MALECHA et R. WISNESKY : Using Dependent Types and Tactics to Enable Semantic Optimization of Language-integrated Queries. *In Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 49–58, 2015.
- [23] C. MARCHÉ : *The Krakatoa Verification Tool for Java programs*. INRIA, 2015.
- [24] THE COQ DEVELOPMENT TEAM : *The Coq proof assistant reference manual*. LogiCal Project, 2016. Version 8.5pl2.
- [25] G. MOERKOTTE : *Building Query Compilers*. Univ. Mannheim, 2014.
- [26] H. PIRAHESH, J. M. HELLERSTEIN et W. HASAN : Extensible/rule based query rewrite optimization in Starburst. *In SIGMOD*, pages 39–48, 1992.
- [27] F. POTTIER et Y. RÉGIS-GIANNAS : Menhir reference manual, 2016.
- [28] A. SHINNAR, J. SIMÉON et M. HIRZEL : A pattern calculus for rule languages: Expressiveness, compilation, and mechanization. *In ECOOP*, 2015.
- [29] V. TANNEN, P. BUNEMAN et L. WONG : Naturally embedded query languages. *In ICDT*, 1992.
- [30] P. W. TRINDER et P. L. WADLER : List comprehensions and the relational calculus. *In GalFp*, 1988.
- [31] J. ULLMAN et J. WIDOM : *Database Systems: The Complete Book*. Prentice Hall, 2000.
- [32] J. VAN DEN BUSSCHE et S. VANSUMMEREN : Polymorphic type inference for the named nested relational calculus. *Transactions on Computational Logic (TOCL)*, 9(1), 2007.
- [33] M. ZAHARIA, M. CHOWDHURY, T. DAS, A. DAVE, J. MA, M. MCCAULEY, M. FRANKLIN, S. SHENKER et I. STOICA : Fast and interactive analytics over hadoop data with spark. *USENIX Login*, 37(4):45–51, 2012.