Rapport de Stage
Vérification de compilation de requêtes SQL à base de traces

Léo Andrès

28 août 2017

Résumé

Ce document est un rapport de mon stage *Vérification de compilation de requêtes SQL à base de traces*, effectué dans le cadre de ma formation en Magistère 1 à l'Université Paris-Sud. Ce stage s'est déroulé sous la direction conjointe de Véronique Benzaken, Évelyne Contejean et Chantal Keller.

Table des matières

1 Le Laboratoire de Recherche en Informatique

Le Laboratoire de Recherche en Informatique (LRI) est une Unité Mixte de Recherche (UMR) associant l'Université Paris-Sud et le CNRS.

Le LRI est composé de plusieurs équipes, parmi lesquelles, l'équipe vérification d'Algorithmes, Langages et systèmes (VALS) au sein de laquelle s'est déroulé ce stage. L'équipe VALS est particulièrement orientée vers la recherche dans le domaine des méthodes formelles.

2 À propos des méthodes formelles

2.1 Trouver l'erreur

Lorsqu'on exécute un programme informatique que l'on vient d'écrire, il est courant que le résultat ne soit pas celui que l'on souhaitait. Le plus souvent, cela est dû à une erreur dans le code source du programme, parfois dans les bibliothèques que l'on utilise, rarement dans le compilateur et presque jamais dans le matériel.

Cela peut s'expliquer en partie par le fait que le compilateur est utilisé par un plus grand nombre d'utilisateurs que les bibliothèques, lesquelles sont elles-mêmes plus utilisées que notre programme.

Faut-il alors attendre qu'un utilisateur de notre programme remarque un bug pour pouvoir le corriger ? Dans le cas des systèmes critiques — transports, énergie, santé etc. — il est impératif de pouvoir détecter les problèmes avant qu'ils ne se manifestent ; dans le cas des autres logiciels, on préfère aussi que cela soit le cas même si ce n'est pas une condition nécessaire.

2.2 Tests manuels et unitaires

Un des moyens de détecter les bugs, est tout simplement de *tester* les programmes. Cela peut se faire à la main, mais cette façon de faire devient très vite fastidieuse dans le cas de gros programmes. On peut donc automatiser les tests. Les tests unitaires sont un exemple de tests automatisés, il s'agit par exemple d'écrire un ensemble de tests pour chacune des fonctions de notre programme. Une fois les tests écrits, on peut ainsi vérifier rapidement que le programme passe les tests, ce qui s'avère pratique lorsque l'on modifie le code du programme par exemple.

Cependant, avec cette méthode, si l'on oublie un cas particulier dans nos tests, on aura l'impression que le programme est correct, alors qu'il ne l'est pas. Prenons par exemple cet algorithme écrit en ocaml qui recherche l'entier maximum d'une liste supposée non-vide :

```
let max_in_list = function
  List.fold_left (fun max el -> if el > max then el else
  ↪  max) 0
```

Supposons que l'on ait écrit les tests suivants pour notre algorithme :

```
(* on suppose que notre framework de test contient une
 ↪  fonction assert_equal qui lève une exception si ses
 ↪  deux paramètres sont différents *)

let test_max_in_list () =

  assert_equal 0 (max_in_list [0]);
  assert_equal 73 (max_in_list [73]);
  assert_equal 3 (max_in_list [0; 1; 2; 3]);
  assert_equal 1357 (max_in_list [549; 0; 1; 2; 64;
   ↪  1035; 1357; 10; 1345])
```

Notre algorithme ne va échouer sur aucun des tests et l'on pourrait alors croire qu'il est correct. Ce n'est bien évidemment pas le cas. Si notre liste n'est composée que de nombres entiers négatifs, l'algorithme renverra 0 dans tous les cas... Le test suivant aurait pu détecter cela :

```
assert_equal -1 (max_in_list [-1]);
```

Une version correcte de l'algorithme serait :

```
let max_in_list = function
  | [x] -> x
  | x::s -> List.fold_left (fun max el -> if el > max
   ↪  then el else max) x s
```

On voit bien alors une des faiblesses des tests : on ne peut jamais être certain d'avoir pensé à tous les cas et on ne peut généralement pas tous les tester lorsqu'il y en a une infinité ou qu'ils sont trop nombreux.

2.3 Un exemple de méthode formelle

Il existe un autre ensemble de façons de procéder, que l'on appelle méthodes formelles. Ces méthodes permettent de vérifier que pour n'importe quel cas, on aura bien le comportement attendu. On peut par exemple exprimer le résultat attendu sous forme de formules logiques, puis, laisser un outil analyser notre programme et nous dire s'il satisfait bien ces formules. Le fait de décrire le comportement attendu sous forme de formules logiques est appelé la *spécification*.

Prenons l'exemple de recherche du maximum dans une liste d'entiers supposée non-vide. Avec l la liste passée à notre fonction et r son résultat, la spécification pourrait être la suivante :

$$\forall x \in l, r \geq x$$

Si l'outil chargé de vérifier que notre programme satisfait bien cette formule nous affirme que la spécification est respectée, on a alors de fortes garanties sur la *correction* de celui-ci, puisqu'en effet, on reste très général sur notre liste : on ne se réduit pas à quelques cas comme précédemment avec les tests.

Cependant, si on y regarde de plus près, on s'aperçoit que le programme suivant satisfait aussi cette formule :

```
let max_in_list = function
  | [x] -> x + 1
  | x::s -> List.fold_left (fun max el -> if el > max
  ↪  then el + 1 else max) (x + 1) s
```

Ici, le problème n'est donc plus celui des cas particuliers, mais celui de la spécification : ici, on a oublié de préciser que la valeur renvoyée par notre fonction doit être présente dans la liste. Cela pourrait s'exprimer ainsi :

$$r \in l$$

Il est intéressant de noter que le fait que l'algorithme précédent est incorrect aurait été détecté très facilement avec un test.

2.4 Les différents aspects des méthodes formelles

Les méthodes formelles ne se limitent pas à la vérification d'une spécification pour un algorithme. On peut aussi prouver des propriétés diverses sur des langages, garantir que les états d'un système respectent des contraintes, démontrer des théorèmes mathématiques. Tout cela implique de nombreux travaux tels que le développement de langages de spécification adaptés à des langages particuliers, de solveurs, la formalisation de la sémantique de langages etc.

3 Travail effectué durant le stage

3.1 Contexte scientifique du stage

Ce stage s'inscrit dans le cadre du projet DataCert de l'ANR. Un des objectifs de ce projet est d'appliquer des méthodes formelles aux différents outils utilisés en informatique traitant principalement des *données*. En effet, jusqu'à présent, peu de travaux impliquant les méthodes formelles ont été effectués sur les deux outils principaux utilisés pour gérer des données en informatique, à savoir, les systèmes de Gestion de Bases de Données (SGBD) et le langage principal utilisé pour communiquer avec eux depuis un autre programme : SQL. Pourtant, le volume de données échangé dans le monde ne cesse de croître.

3.2 SQL

Le langage SQL a été le point de départ du stage. Il a tout d'abord fallu écrire un *parser* SQL en OCaml, tout d'abord avec OCamllex et Menhir.

Ensuite, il a suffi de modifier quelques lignes pour pouvoir utiliser le générateur de parser certifié intégré à Menhir, utilisé dans CompCert, lequel a été initialement écrit par Jacques-Henri Jourdan. Nous avons ainsi obtenu automatiquement un parser en Coq avec la preuve de correction — si le parser réussit, alors la phrase parsée fait partie de la grammaire — et la preuve de complétude — si la grammaire n'est pas ambiguë, alors, si une phrase fait partie de la grammaire, le parser doit réussir.

Finalement, grâce à un code OCaml, il a été possible de modifier automatiquement l'AST obtenu pour qu'il corresponde à la syntaxe utilisée dans SQLCoq, une bibliothèque formalisant la sémantique de SQL en Coq.

3.3 Les plans d'exécution

Interviennent alors les *plans d'exécution*. En effet, SQL étant un langage déclaratif — on décrit ce que l'on veut obtenir et non pas comment l'obtenir, contrairement à des langages tels que C, Java ou OCaml — il est généralement utilisé conjointement avec un SGBD. Pour une requête SQL donnée, c'est le SGBD qui va décider des algorithmes à utiliser pour aller récupérer sur le disque les données qui nous intéressent. On peut demander au SGBD de nous fournir un plan d'exécution de la requête, c'est-à-dire de nous indiquer les algorithmes qu'il va utiliser pour exécuter la requête. Par exemple, pour la requête SQL suivante :

```
SELECT mid, title FROM movie WHERE mid > 2500;
```

On pourrait obtenir un plan comme celui-ci :

```
<Plan>
  <Node>        Seq Scan        </Node>
  <Relation>    movie           </Relation>
  <Projection>  mid, title      </Projection>
  <Filter>      (mid &gt; 2500) </Filter>
</Plan>
```

Cependant, il n'existe pas de standard portant sur les plans d'exécution, les SGBD les présentent de façon différente, utilisent leurs propres algorithmes, utilisent un nom différent pour un même algorithme, ne donnent pas la même quantité d'informations etc.

Après avoir étudié quelques plans fournis par quatre SGBD, à savoir MySQL, SQLite, Oracle Database et PostgreSQL, nous avons remarqué que seuls deux d'entre

eux fournissaient des plans suffisamment complets pour pouvoir être utilisés ensuite, il s'agit d'Oracle Database et de PostgreSQL.

J'ai alors dû écrire un parser pour les plans d'exécution fournis par PostgreSQL tandis qu'une autre personne faisait la même chose pour Oracle Database. Ensuite, je me suis chargé de définir un format *unifié* permettant de décrire les plans fournis par ces deux SGBD de la même façon. Il a ensuite fallu écrire deux programmes permettant de passer des AST obtenus en parsant les plans à ce nouveau format unifié ; ainsi qu'un parser pour les nouveaux plans respectant ce format unifié.

Ainsi, à partir d'une même requête SQL, il est possible de la faire traiter par deux SGBD différents et d'obtenir deux plans d'exécution, parfois différents, mais respectant une syntaxe commune de description des plans.

3.4 ExtAlg

La sémantique d'une requête SQL peut être exprimée au moyen de l'algèbre relationnelle. Avec SQLCoq, il est possible de relier une requête SQL non pas à son équivalent en algèbre relationnelle, mais à son équivalent dans une algèbre étendue, appelée ExtAlg. Ainsi, pour pouvoir vérifier qu'un SGBD, lorsqu'il exécute une requête, préserve la sémantique de celle-ci, il fallait pouvoir comparer le plan engendré par celle-ci à son équivalent en ExtAlg. J'ai donc écrit un programme OCaml qui, à partir d'un plan, produit une expression d'ExtAlg lui correspondant. Dès lors, si l'on arrive à montrer l'équivalence entre l'expression fournie par SQLCoq à partir de la requête et l'expression obtenue à partir du plan fourni par le SGBD, on sait que le SGBD a bien préservé la sémantique de la requête. Un exemple de requête SQL, du plan obtenu à partir de celle-ci et d'expression algébrique obtenue à partir du plan est donné dans l'annexe exemple complet.

ExtAlg, tout comme l'algèbre relationnelle, manipule des relations, sur lesquelles on peut appliquer notamment le produit cartésien et l'opérateur ω. L'opérateur ω prend en compte quatre paramètres : les éventuelles partitions, la projection et le renommage, les conditions de sélection et enfin, la relation sur laquelle on l'applique. Il est nécessaire d'ajouter des variables contenant une requête — requête qui n'est pas affichée en sortie pour des raisons de lisibilité — nommées x_i. De plus, ici, la requête est annotée avec des *labels* et les noms des algorithmes venant du plan, afin de pouvoir plus tard, simuler l'exécution au moyen d'algorithmes écrits en Coq.

Il faut noter que certains SGBD ne fournissent pas toutes les informations nécessaires, PostgreSQL, par exemple, ne fournit aucune information sur l'équivalent des *projections* dans les plans, ainsi, il a fallu être capable de retrouver l'information à partir de la requête originale, en attendant que le code de PostgreSQL soit modifié afin d'ajouter ces informations aux plans.

Finalement, il restait à prouver l'équivalence entre deux requêtes d'ExtAlg. Pour cela, j'ai commencé l'écriture d'un programme OCaml qui normalise une requête donnée, point de départ nécessaire pour pouvoir prouver l'équivalence de deux requêtes par d'autres équivalences connues et démontrées.

4 Leçons

Ce stage m'a permis d'approfondir mes connaissances dans plusieurs domaines. Tout d'abord, l'écriture de parser, chose que je n'avais jamais réalisée avant. De plus, j'ai eu l'occasion d'approfondir mes connaissances du langage SQL, des différents SGBD, mais surtout, de la façon dont ces derniers fonctionnent : les algorithmes courants, leur lien avec l'algèbre relationnelle etc.

Au-delà de l'aspect informatique, cela a surtout été l'occasion de découvrir le monde de la recherche et d'être, grâce à cela, sûr, dorénavant, de vouloir m'orienter vers la recherche. En effet, cela a été à la fois très enrichissant et stimulant. Pour arriver à résoudre les problèmes que l'on rencontre, il n'est plus question de procéder comme lorsque l'on suit un TP en cours d'informatique ; il n'y a plus une liste linéaire de questions intermédiaires nous guidant directement au résultat et qu'il suffit de résoudre une par une. Il s'agit plutôt de comprendre suffisamment le problème afin de trouver soi-même cette liste d'étapes que l'on va devoir franchir pour arriver au résultat souhaité, sans garantie sur le fait de partir dans la bonne direction. On se trouve alors dans une situation où l'on effectue un travail conscient, technique, dans une direction jusqu'à rencontrer un nouveau problème ou réaliser que cette direction n'était pas la bonne. Parfois, aucune solution n'apparaît clairement, jusqu'à ce que quelques heures, voire quelques jours plus tard, elle nous apparaisse subitement ; une autre possibilité est la discussion du problème avec les autres chercheurs, chose très courante d'après mon expérience de stage, car souvent, une autre personne envisagera le problème d'une autre façon et saura trouver comment le contourner. D'autre part, l'échange avec les autres chercheurs est toujours présent, même avec ceux avec qui l'on ne travaille pas directement, permettant ainsi de toujours avoir l'occasion de réfléchir sur nombre de sujets en permanence. Tout cela rend le travail quotidien très intéressant, agréable et est par la même occasion très formateur. Ainsi, je tiens à remercier Véronique Benzaken, Évelyne Contejean et Chantal Keller pour m'avoir confié des problèmes intéressants, d'avoir pris le temps de m'expliquer les problématiques en profondeur, de m'avoir toujours guidé lorsque cela était nécessaire, mais surtout, de s'être toujours montrées très compréhensives et amicales, rendant par là ma première expérience de recherche extrêmement satisfaisante.

A Exemple complet

```
select s.sname from sailors s where exists (select *
 ↪  from reserves r where r.bid = 103 and s.sid = r.sid);
```

```
<?xml version="1.0"?>
<explain>
  <Query>
    <Plan>
      <Node-Type>Hash Join</Node-Type>
      <Join-Type>Inner</Join-Type>
      <Hash-Cond>(s.sid = r.sid)</Hash-Cond>
      <Plans>
        <Plan>
          <Node-Type>Seq Scan</Node-Type>
          <Parent-Relationship>Outer</Parent-Relationshi
           ↪  p>
          <Relation-Name>sailors</Relation-Name>
          <Alias>s</Alias>
        </Plan>
        <Plan>
          <Node-Type>Hash</Node-Type>
          <Parent-Relationship>Inner</Parent-Relationshi
           ↪  p>
          <Plans>
            <Plan>
              <Node-Type>Aggregate</Node-Type>
              <Strategy>Hashed</Strategy>
              <Parent-Relationship>Outer</Parent-Relatio
               ↪  nship>
              <Group-Key>
                <Item>r.sid</Item>
              </Group-Key>
              <Plans>
                <Plan>
                  <Node-Type>Seq Scan</Node-Type>
                  <Parent-Relationship>Outer</Parent-Rel
                   ↪  ationship>
                  <Relation-Name>reserves</Relation-Name>
                  <Alias>r</Alias>
                  <Filter>(bid = 103)</Filter>
                </Plan>
              </Plans>
            </Plan>
          </Plans>
        </Plan>
```

```
      </Plans>
    </Plan>
  </Query>
</explain>
```

```
ω {
  _ Fine
  _ r.dday → r.dday, r.bid → r.bid, r.sid → r.sid,
    ↪  s.age → s.age, s.rating → s.rating, s.sname
    ↪  → s.sname, s.sid → s.sid
  _ x_0.s.sid = x_0.r.sid
  _ {
      ω {
        _ Fine
        _ sid → s.sid, sname → s.sname, rating →
          ↪  s.rating, age → s.age
        _ T
        _ {
            Seq Scan on sailors
        }
      }
      × (DependentJoin) (Hash Join)
      Label : Hash
      ω {
        _ Partition (x_0.r.sid)
        _ r.sid → r.sid, r.bid → r.bid, r.dday →
          ↪  r.dday
        _ T
        _ {
          Label : Aggregate
          ω {
            _ Fine
            _ sid → r.sid, bid → r.bid, dday →
              ↪  r.dday
            _ x_0.bid = 103
            _ {
                Seq Scan on reserves
            }
          }
        }
      }
  }
}
```

La ligne 27 indique un *Seq Scan* sur la relation *reserves*, si l'on enlève l'indication de l'algorithme utilisé, cela revient à simplement parler de la relation *reserves*.

Sur les lignes 22 à 29, on utilise ω sur cette relation avec des paramètres particuliers. La ligne 23 nous indique qu'il n'y a aucune partition, ce qui se note *Fine*. La ligne 24 nous indique qu'on va sélectionner les attributs *sid*, *bid* et *dday* de la relation, et qu'on les renomme respectivement en *r.sid*, *r.bid* et *r.dday*. La ligne 25 indique quant à elle que l'on va filtrer le résultat en ne gardant que les lignes respectant la condition $\texttt{bid} = 103$. L'écriture mathématique de cette expression, en faisant abstraction des variables notées x_i, serait la suivante :

$$\omega_{\text{Fine,r.sid} \to \text{r.sid, r.bid} \to \text{r.bid, r.dday} \to \text{r.dday,bid=103}}(\text{reserves})$$

Les variables notées x_i servent à conserver des informations parfois nécessaires ailleurs alors qu'elles ne sont pas disponible. En effet, le SGBD, afin d'optimiser l'exécution des requêtes, va parfois appliquer un filtre en fonction de la valeur d'un attribut qui n'est pas disponible sur les relations qu'il est en train de traiter, il faut donc enregistrer certaines informations pour pouvoir retrouver ces valeurs.

On peut aussi noter qu'à la ligne 14, il y a : *× (DependentJoin) (Hash Join)*, *Hash Join* est simplement l'algorithme qui va être utilisé, il ne fait pas partie de l'algèbre étendue. Cependant, il est intéressant de noter que le *Join* est dit *Dependent*, en effet, là aussi, le SGBD va souvent avoir besoin du résultat d'une des deux relations composant la jointure pour pouvoir calculer l'autre. Ainsi, on note par défaut toutes les jointures comme étant des *DependentJoin* et ce n'est que lors de la phase de normalisation que l'on détectera si le *Dependent* est vraiment nécessaire ou non — ici, ça n'est par exemple pas le cas.