A Coq mechanised formal semantics for realistic SQL queries : Formally reconciling SQL and (extended) relational algebra.

Véronique Benzaken and Évelyne Contejean

August 30, 2018

A Coq mechanised formal semantics for realistic SQL queries*

Formally reconciling SQL and bag relational algebra

V. Benzaken
Université Paris Sud, LRI
veronique.benzaken@lri.fr

É. Contejean
CNRS, Université Paris Sud, LRI
evelyne.contejean@lri.fr

Abstract

In this article, we provide a Coq mechanised, executable, formal semantics for realistic SQL queries consisting of `select [distinct] from where group by having` queries with `NULL` *values, functions, aggregates, quantifiers* and *nested* potentially *correlated* sub-queries. We then relate this fragment to a Coq formalised (extended) relational algebra that enjoys a *bag* semantics. Doing so we provide the first *formally mechanised proof* of the *equivalence* of SQL and extended relational algebra and, from a compilation perspective, thanks to the Coq extraction mechanism to Ocaml, a Coq certified semantic analyser for a SQL compiler.

ACM Reference format:
V. Benzaken and É. Contejean. 2016. A Coq mechanised formal semantics for realistic SQL queries. In *Proceedings of Université de Paris Sud, Orsay, France, July 2018 (Research Report),* 12 pages.
DOI: 10.475/123_4

1 Introduction

In the area of programming languages, providing a formal semantics for a language is a tricky but crucial task as it allows compilers to rigorously reason about program behaviours and to verify the correctness of optimisations [17, 23]. When considering real-life programming languages the task becomes even harder as it happens that the specifications of the language are often written in natural language. Even when they are formal, they only account for a limited subset of the considered language and are, most of the time, human-checked proven correct. In all cases, there are few strong guarantees that the whole faithfully accounts for the exact semantics and correctness of performed optimisations. To obtain such high level guarantees, a promising approach consists in using *proof assistants* such as Coq [25] or Isabelle [26] to define *mechanised*, *executable* semantics whose correctness is *machine-checkable.*

A shining demonstration of the viability of this approach for real systems is Leroy's CompCert project [20], which specified, implemented, and proved the correctness of an optimising C compiler. This compiler is not a toy: it compiles essentially the whole ISO C99 language, targets several architectures, and achieves 90% of the performance of GCC's optimisation level 1. The value of CompCert's correctness proofs has surprised some observers. Quoting [27] that used random testing to assess all popular C compilers: *"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. [...]. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users."*

Our long-term goal, based on the same approach, aims at providing a Coq verified compiler for SQL, *the* standard in terms of programming languages for relational database systems. In this article we focus on semantical issues and define, using Coq, a formal semantics for SQL. More precisely, we define SQL_{Coq} (syntax and semantics), a Coq formalisation of SQL accounting for `select [distinct] from where group by having` queries with `NULL` *values, functions, aggregates, quantifiers* and *nested* potentially *correlated* sub-queries. In order to convince ourselves that SQL_{Coq} fairly reflects SQL's semantics we developped a query generator that serves to automatically generate and execute queries against mainstream systems such as Postgresql, Oracle™, and also against SQL_{Coq}. Obtaining the same results we hence give strong credibility of the (semantic) relevance of SQL_{Coq}. Doing so, we provide a Coq certified semantic analyser for the compiler. We then

*Work funded by the DataCert ANR project: ANR-15-CE39-0009.

Research Report, Orsay, France
© 2016 Copyright held by the owner/author(s). 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

formalise, using Coq, SQL_{Alg}, a *bag*, *environment-aware* relational algebra similar to, while extending it, the one presented in [12]. Next, we formally relate, through a Coq mechanised translation, SQL_{Coq} to SQL_{Alg} and vice-versa. By proving an adequacy theorem stating that these translations preserve semantics, we provide the first, to our best knowledge, *mechanised*, formal proof of equivalence between the considered SQL fragment and a bag relationnal algebra.

Related works Many attempts have been made by the database community to define a formal semantics for SQL. Among those, proposals can be found in [7, 22] where the authors presented a translation from SQL to an extended algebra. A credible subset of SQL (with no functions symbols, nested queries, `NULL`'s nor bags though) was addressed. The first formal semantics for SQL accounting for `NULL`'s and bags is found in [16]. However, the work did not consider `group by having` clauses, aggregates nor *complex* expressions: their expressions consist of attributes' names or constants. As will be shown in Section 2 and 3, the treatment of complex expressions is very subtle.

On the proof assistant side, the first attempt to formalise the (unnamed version of) the relational data model, using the Agda proof assistant [24], is found in [13, 14] while the first, almost complete, Coq formalisation of the relational model is found in [4] where the data model, algebra, tableaux queries, the chase as well as integrity constraints aspects were modelled. A convincing mechanisation, based on nominal Isabelle, of a subset of XQuery [19] is given in [8]. Recently, an SSreflect-based mechanisation of the Datalog language has been proposed in [5]

The very first attempt to verify, using Coq, a RDBMS is presented in [21]. However the SQL fragment they addressed was rather unrealistic as, probably for the sake of simplicity, they placed themselves in the context of an unnamed version of the language in which attributes were denoted by positions. They did not consider `group by having` clause, neither `NULL`'s nor aggregates.

More recently, a Coq modelisation of the nested relational algebra (NRA [10]) which directly serves as a semantics for SQL is provided in [2]. Finally, the closest proposal in terms of mechanised semantics for SQL is addressed in [9]. The authors describe a tool to decide whether two SQL queries are equivalent. To do so, they defined HottSQL, a K-relation [15] based semantics for SQL. Unlike ours, the considered fragment does not handle `NULL` values nor `having` clause and they used a reconstruction of the language thus not accounting for the trickier aspects of variable binding. As we shall see in Section 2 and 3 the treatment of attributes' names and more generally environments is particularly a tough task. Furthermore, they relaxed the *finite support* constraint imposed to K-relations hence possibly yielding *infinite* query results as well as potentially *infinitely many* occurences of tuples in queries' results. Last, and more importantly, their semantics is not *executable* hence it is impossible to verify whether they do implement the correct SQL's semantics.

Unlike those works we propose (i) a bag *mechanised executable* semantics for the realistic subset of SQL previously mentionned. (ii) By relating this fragment to a Coq formalised relational algebra that enjoys a *bag* semantics we provide (iii) the first *formally mechanised proof* of the *equivalence* of SQL and algebra and, from a compilation perspective, thanks to the Coq extraction mechanism to Ocaml, (iv) a *Coq certified semantic analyser* for a SQL compiler.

Organisation In Section 2, we first present SQL's subtleties that are mandatory to be taken into account to provide a realistic semantics. Then we detail, in Section 3, SQL_{Coq}'s syntax and semantics and comment on our experimental assessment thanks to our random query generator. Section 4 is devoted to the mechanisation of SQL_{Alg}, the bag algebra. Then the translations between SQL_{Coq} and SQL_{Alg} as well as the equivalence theorem are presented. We conclude, draw lessons and give perpectives in Section 5.

2 SQL: simple and subtle

SQL is a declarative language. As such it is often considered simple. However, its semantics is more subtle than appears at first sight. SQL's semantics is described by the ISO Standard [18] which consists of thousand pages written in natural language. It is often unclear and, thus, cannot serve as a formal semantics. This explains why many vendors implement various aspects of it in their own way as witnessed by [1]. Although the Standard cannot serve as a formal specification, we relied on it and, meanwhile, we tested our development against systems like Postgresql and Oracle™ to better grasp SQL's semantics.

It is well known that SQL's `select from where` construct enjoys a bag semantics: the same tuple can occur several times in the result. Purely set-theoretic operators such as ∪ (`union`), ∩ (`intersect`) and \ (`except`) have a set-theoretic one. Therefore any formal semantics must account for both sets and bags. As pointed out in Guagliardo *et al.*, in [16], SQL deals with `NULL` values that are intended to represent unknown information. A 3-valued logic combined with the classical Boolean logic is used to handle them (even if 3-valued logic is not necessary as shown in [16] but in the *absence of quantifiers*). However `NULL`'s are not treated in a uniform way according to the context as illustrated in Section 2.1. Last, and

r
a
1
NULL

s
a
NULL

t
a
1
NULL
NULL

t_1											
a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1
1	1	1	6	2	1	2	6	3	1	4	6
1	2	1	7	2	2	2	7	3	2	4	7
1	3	1	8	2	3	2	8	3	3	4	8
1	4	1	9	2	4	2	9	3	4	4	9
1	5	1	10	2	5	2	10	3	5	4	10

t_2	
a_2	b_2
7	7
7	7

```
-- q1
select r.a from r where r.a not in (select s.a from s);
-- q2
select r.a from r where
   not exists (select * from s where s.a = r.a);
-- q3
select r.a from r except select s.a from s;
-- q4
select t.a,count(*) as c from t group by t.a;
-- q5
select a1, max(b1) from t1 group by a1;
-- q6
select a1 from t1 group by a1 having
exists (select a2 from t2 group by a2 having sum(1+0*a1) = 10);
-- q7
select a1 from t1 group by a1 having
exists (select a2 from t2 group by a2 having sum(1+0*a2) = 10);
-- q8
select a1 from t1 group by a1 having
exists (select a2 from t2 group by a2 having sum(1+0*a2) = 2);
```

```
-- q9(k)
select a1 from t1 group by a1 having
exists (select a2 from t2 group by a2 having sum(1) = k);
-- q10
select a1 from t1 group by a1 having
exists (select a2 from t2
             group by a2 having sum(1+0*a1)+sum(1+0*a2) = 12);
-- q11(k)
select a1 from t1 group by a1 having
exists (select a2 from t2 group by a2 having sum(1+0*a1+0*a2) = k);
-- q12
select a1 from t1 group by a1 having
exists (select a2 from t2 group by a2 having sum(1+0*b1+0*b2) = 10);
-- q13(k)
select a1 from t1 group by a1 having
exists (select a2 from t2 group by a2 having sum(1+0*a1+0*b2) = k);
-- q14
select a1 from t1 group by a1 having
exists (select a2 from t2 group by a2 having sum(1+0*b1+0*a2) = 12);
```

Figure 1. Semantically subtle queries.

more importantly, as illustrated in Section 2.2, the way SQL manages environments and expressions is complex and represent the most tricky aspect to address in order to obtain a formal semantics for realistic SQL queries. In the sequel of this section, we note ⟦q⟧ the result of the evaluation of query q, (), the tuple constructor, [ ] the list constructor, { } the set constructor and ⦃ ⦄ the bag constructor. Figure 1 gathers a bunch of queries that will illustrate SQL's most subtle aspects.

2.1 NULL values

The three first queries are borrowed from [16]. They examplify the fact that NULL is neither equal to nor different from any other value (including itself): comparing NULL with any expression always yields unknown. Query q1 returns an empty result. This is explained by the fact that ⟦select s.a from s⟧ = ⦃(s.a=NULL)⦄, hence over all tuples (r.a=x), in particular over (r.a=1) and (r.a=NULL), ⟦r.a not in select s.a from s⟧ yields not unknown, that is unknown, which is eventually considered as false, as remarked by Guagliardo *et al.*, in [16]. Neither tuple belongs to the result of the first query.

Query q2 returns ⦃(r.a=1); (r.a=NULL)⦄. Let subq2 be (select * from s where s.a = r.a), it yields an empty result over all tuples (r.a=x), hence ⟦exists (subq2)⟧ is always false and ⟦not exists (subq2)⟧ is always true, thus (r.a=1) and (r.a=NULL) are in the result of q2.

Query q3 returns {(r.a=1)}, because the set difference does not use 3-valued logical equality, but standard syntactic equality. Here both tuples (r.a=NULL) and (s.a=NULL) are equal.

Last, query q4 returns ⦃(t.a=NULL,c=2); (t.a=1,c=1)⦄. This illustrates the fact that NULL, which is neither equal nor different from NULL in a 3-valued logic, is indeed equal to NULL in the context of grouping. The semantics proposed in Section 3.2 will account for such behaviours.

2.2 Expressions in environments

Let us now address the way SQL manages evaluation environments in presence of aggregates and nested correlated queries. In order to evaluate simple (without aggregates) expressions, it is enough to have a single environment, containing information about the bounded attributes and the values for them. In this simple case (*e.g.*, select a1, b1 from t1;) such an environment corresponds to a unique tuple (a1=x,b1=y) where x and y range in the active domains of a1 and b1 respectively.

Evaluating expressions with aggregates is more involved, since an aggregate operates over a list of values, each one corresponding to a tuple. The crucial point is to understand how such a list of tuples is produced. Section 10.9 of [18] (< aggregate functions >, *how to retrieve the rows* – page 545) should provide some guidance in answering this question. Unfortunately it was of no help. We thus proceeded by testing many queries over Postgresql and Oracle™. It took us significant effort to reach the *semantically relevant* set of queries which are given in Figure 1. For all of them, we obtained the same results on both systems. Let us comment on these queries. For `q5` the result is:

$$\left\{\!\left|\ \begin{array}{c}(\texttt{a1=1,max=10});(\texttt{a1=2,max=10});\\(\texttt{a1=3,max=5});(\texttt{a1=4,max=10})\end{array}\ \right|\!\right\}$$

It is easy to understand what happens when evaluating `max(b1)` in `q5`: each group (where `a1` is fixed) contains some tuples, each of them yielding a value for `b1`. Then `max` is computed over this list of values. For instance, the group T_1 where `a1=1` contains exactly one occurence of tuples of the form `(a1=1,b1=i)`, where `i` ranges from 1 to 10, hence `b1` ranges from 1 to 10, and `max(b1)` is equal to 10, whereas the group where `a1=3` contains tuples `(a1=3,b1=i)`, where `i=1,...,5`, and `max(b1)` is equal to 5. In this simple case a group of n tuples merely yields n simple environments each of them consisting of a single tuple – we say that the group has been *split into* individual tuples.

The situation gets more complex when evaluating an aggregate expression in a nested sub-query. How to build, in that case, the suitable list of environments (tuples) in order to get the needed list of values as arguments of the aggregate itself? Assuming that an aggregate expression occurs in a subquery under more than two grouping levels, there are several groups in the evaluation context. How to combine them in order to obtain the correct list of tuples? Which groups have to be split into and which have not to? Queries `q6,q7,q8,q9,q10,q11,q12,q13` and table `t2` have been designed to answer these questions.

Query `q6`'s result is $\{\!|(\texttt{a1=1});(\texttt{a1=2})|\!\}$. This means that subquery `select a2 from t2 group by a2 having sum(1+0*a1)=10` is not empty when `a1=1`, `a1=2`, and is empty when `a1=3` and `a1=4`. For `a1=1`, this subquery is evaluted under the context of group T_1 seen above for `q5`. This indicates that expression `sum(1+0*a1)` is evaluated to 10 in the context of both outer group T_1 and inner group $T_2 = \{\!|(\texttt{a2=7,b2=7});(\texttt{a2=7,b2=7})|\!\}$. Hence, the relevant evaluation context of `sum(1+0*a1)` in $[T_2;T_1]$ has to contain 10 tuples, each of them contributing by `1+0*a1`, that is 1, to the `sum`. A simple reasoning about groups' cardinality allows to conclude that group T_1 has been split into, and not T_2.

Let us now consider `q7`. While very similar to `q6`, except that the `sum` is over `1+0*a2`, $[\![\texttt{q7}]\!]$ is empty, meaning that the evaluation context of `sum(1+0*a2)` in $[T_2;T_1]$ does not to contain 10 tuples. How many tuples does it actually contain? An educated guess is 2 (that is T_2's cardinality), which is confirmed by the fact that $[\![\texttt{q8}]\!]$ indeed contains `(a1=1)`.

So, in the same context, $[T_2;T_1]$, SQL computes 10 values for `1+0*a1`, from 10 tuples, and 2 values for `1+0*a2`, from 2 tuples. The only sensible solution is that the values, and their corresponding tuples depend not only from the group context, but also from the expression to be evaluated. Given a context and an expression, there is a single relevant group, which is split into: T_1 for `1+0*a1`, and T_2 for `1+0*a2` in the context $[T_2;T_1]$.

Another interesting case is when the expression under the aggregate is a constant value k, as in `q9(`k`)`. What should be the relevant group to be split into? Is there even such a relevant group corresponding to the set of all attributes of `sum(1)`, that is the empty set?

Actually `q9(2)` yields the same result as `q8`, meaning that the relevant group for a constant is the innermost group T_2. Surprisingly, compared to `q7`, `q9(10)` is empty, which means that usual arithmetic equalities, such as $1{+}0{*}a1 = 1$ are no longer valid in SQL, under aggregates.

At that point, what happens if both expressions `1+0*a1` and `1+0*a2` have to be evaluated in the same group context as it is the case for `q10`, where `1+0*a1` and `1+0*a2` occur under *distinct* aggregates. There is no single obvious relevant group anymore. `q10`'s result contains `(a1=1)`, meaning that both expressions `1+0*a1` and `1+0*a2` have been evaluated *independently*, the first in a context where T_1 has been split into, and the second where the splitted group is T_2. This makes clear that *SQL allows two sub-expressions of a given expression to be evaluated in different contexts* which is definitely contrary to what is done in other mainstream programming languages!

What if `1+0*a1` and `1+0*a2` occur under the *same* aggregate, as in `q11(`k`)`? When k=2, $[\![\texttt{q11(2)}]\!]$ is $\bigcup_{i=1}^{i=4} \{\!|(\texttt{a1=i})|\!\}$, otherwise $[\![\texttt{q11(}k\texttt{)}]\!]$ is empty. Therefore T_2, the innermost relevant group, has been split into.

As the reader may have noticed, all expressions under the aggregates were built upon grouping attributes. What happens when such is not the case? Query `q12` contains `sum(1+0*b1+0*b2)` and is not *well formed* according to the Standard, thus, is not evaluated. Next query `q13(`k`)` contains `sum(1+0*a1+0*b2)` and behaves exactly the same as `q11(`k`)` does. The last query, `q14`, which contains `sum(1+0*b1+0*a2)`, is ill-formed and not evaluated.

At that point, we are able to sum up the above lessons and precisely explain how SQL manages environments.

First, when evaluating an expression with aggregates where the top operand is a function (for instance `+`, as in `q10`), each argument is evaluated separately.

Second, when evaluating an expression $\mathfrak{ag}(e)$ where the top operand $\mathfrak{ag}$ is an aggregate, this aggregate is evaluated against a list of values, each of them coming from the evaluation of e over a tuple. The subtle point is to understand how to build the corresponding list of tuples. Let us introduce a few definitions that will be helpful.

An environment, $\mathcal{E}{=}[S_n;\ldots;S_1]$, is a stack of *slices*: one slice per *nesting level* i, the innermost level being on the top. When necessary, we shall equally adopt the following, OCaml-like, notation for environments $\mathcal{E} = (A, G, T) :: \mathcal{E}'$ in order to highlight the list's head. Slices are of the form $S = (A, G, T)$, where A (also noted $A(S)$) contains the relevant attributes for that level of nesting, *i.e.*, the names introduced in the subquery at this level[1]; G the grouping expressions appearing in the `group by` (also noted $G(S)$); and T a non empty list of tuples[2] (also noted $T(S)$).

When e is a constant expression, the list of tuples $T(S_n)$ comes from the innermost slice of environment $\mathcal{E} = [S_n; ...; S_1]$. In the simple case where all attributes of e are introduced at the same level i, the relevant list is simply $T(S_i)$. Otherwise, when attributes of e belong to at least two different levels among $i_k \dots i_1$ where $i_{j+1} > i_j$, there are two cases:

- either the expression is not well-formed (*cf* `q12` and `q14`), because e contains an homogeneous expression of level $i_j, j < k$ which is not grouped.
- or the expression e is exactly built upon the attributes corresponding to the kth level i_k and the grouping expressions[3] of outermost levels $i_{k-1} \dots i_1$. In this case, let t_{i_j} be a fixed tuple chosen in each $T(S_{i_j})$ for $j < k$, then the list of relevant tuples is made of $t \bowtie t_{i_{k-1}} \bowtie \dots \bowtie t_{i_1}$, where t ranges over $T(S_{i_k})$.

We are now able to present our Coq mechanised formal semantics for a realistic fragment of SQL.

3 A formal Coq mechanised semantics for SQL

SQL_{Coq} addresses the fragment consisting of `select [distinct] from where group by having` queries with `NULL` *values, functions, aggregates, quantifiers* and *nested* potentially *correlated* (in `from`, `where` and `having` clauses) sub-queries. It accounts for `in`, `any`, `all` and `exists` constructs and assigns queries a Coq mechanised (bag) semantics that complies with the Standard.

3.1 SQL_{Coq}: syntax

SQL_{Coq}'s syntax is given on Figure 2, Figure 3 and Figure 4 where the left part of figures represents SQL's abstract syntax and the right part the corresponding Coq syntax. We assume that we are given attributes, functions and aggregates. We shall allow strings, integers and booleans to be values, as well as the special `NULL`. On the top of them, we define usual expressions, first without aggregates e^f, and then with aggregates e^a. SQL formulas are similar to first order formulas except they are always interpreted in a finite domain, which is syntactically refered to as `dom` in Figure 3. Such formulas will then be used in the context of SQL_{Alg}.

SQL_{Coq} sticks, syntactically, as much as possible, to SQL's syntax but the SQL-aware reader shall notice that SQL_{Coq} slightly differs from SQL in different ways. First, for the sake of uniformity, we impose to have the whole `select from where group by having` construct (no optional `where` and `group by having` clauses). When the `where` clause is empty, it is forced to `true`. Similarly, as the `group by` clause partitions the collection of tuples obtained evaluating the `from` clause, when no `group by` is present in SQL, we force SQL_{Coq} to work with the finest partition[4] which corresponds to the `Group_Fine` case. We also force explicit and mandatory renaming of attributes, when $*$ is not used. In our syntax, `select a, b from t;` is expressed by `select a as a, b as b from (table t[*]) where true group by Group_Fine having true`. A further, more subtle, point worth to mention is the distinction we make between e^f and e^a. Both are expressions but the former are built only with functions ($\mathfrak{fn}$) and are evaluated on *tuples* while the latter also allow unested[5] aggregates ($\mathfrak{ag}$) and are, in that case, evaluated on *collections of tuples*. In the same line, we used the same language for formulae either occurring in the `where` (dealing with a single tuple) or in the `having` clause (dealing with collections of tuples) simply by identifying each tuple with its corresponding singleton. Also, no aliases for queries are allowed.

3.2 SQL_{Coq}: semantics

Given a tuple t we note $\ell(t)$ the attributes occuring in t. We assume that we are given a database instance $[\![\_]\!]_{db}$ defined as a function from relation names to *bags* of tuples[6] as well as predefined, fixed interpretations, $[\![\_]\!]_p$, for predicates $\mathfrak{pr}$[7], *i.e.*, a function from vectors of values to Booleans, $[\![\_]\!]_a$ and $[\![\_]\!]_f$ for aggregates $\mathfrak{ag}$ and functions $\mathfrak{fn}$ respectively[8]. As established in Section 2, (complex) expressions occuring in (possibly correlated sub-) queries, are evaluated under a sliced environment, $\mathcal{E} = [S_n; ...; S_1]$ (or $\mathcal{E} = (A, G, T) :: \mathcal{E}'$), the innermost level, n, corresponding to the first slice. The evaluation of a syntactic entity e of type x in environment $\mathcal{E}$ will be denoted by $[\![e]\!]^{\mathrm{x}}_{\mathcal{E}}$ (where x is f for expressions built

[1] if this subquery is a `select from ...` these are the names in the `select`.

[2] When there is a grouping clause at this level, it is an homogeneous group, otherwise it is a single tuple.

[3] those appearing in the `group by` clause of the level ; when there are no such grouping expressions, all attributes of the level are allowed.

[4] The partition consisting of the collection of singletons, one singleton for each tuple in the result of the `from`

[5] e^a is of the form: `avg(a)`; `sum(a+b)`; `sum(a+b)+3`; `sum(a+b)+avg(c+3)` but not of `avg(sum(c)+a)`

[6] these multisets enjoy some list-like operators such as `empty`, `map`, `filter`, etc.

[7] $\mathfrak{pr}$ is $<$, `in` etc.

[8] a may be `sum`, `count` etc. and f: `+,*,-` etc.

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

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

Figure 2. Expressions.

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

```
Inductive conjunct : Type := And | Or.
Inductive quantifier : Type := All | Any.

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

Inductive formula (dom : Type): Type :=
  | Conj : conjunct → formula dom → formula dom → formula dom
  | Not : formula dom → formula dom
  | True :  formula dom| Pred :  predicate -> list aggterm -> formula
      dom| Quant :  list aggterm -> predicate -> quantifier -> dom ->
      formula dom| In :  list select -> dom -> formula dom| Exists :
      dom -> formula dom.
```

Figure 3. Formulas, parameterized by a finite domain of interpretation `dom`.

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

```
Inductive select_item : Type :=
  Select_Star | Select_List : list select → select_item.

Inductive att_renaming : Type :=
   | Att_As : attribute → attribute → att_renaming.
Inductive att_renaming_item : Type :=
   | Att_Ren_Star
   | Att_Ren_List : list att_renaming → att_renaming_item.

Inductive group_by : Type :=
   Group_Fine | Group_By : list funterm → group_by.

Inductive set_op : Type := Union | Intersect | Except.

Inductive query : Type :=
  | Table : relname → query
  | Set : set_op → query → query → query
  | Select :
      (** select *) select_item  →
      (** from *) list from_item →
      (** where *) formula query →
      (** group by *) group_by →
      (** having *) formula query → query

with from_item : Type :=
  | From_Item : query → att_renaming_item → sql_from_item.
```

Figure 4. SQL and SQL_{Coq} syntax

only with functions, a for expressions built also with aggregates, b for formulas and q for queries).

The semantics of simple expressions, which poses no difficulties, is given in Figure 5. The semantics of complex expressions detailed in Figure 6, deserves comments. When the complex expression is headed by a function, $\mathfrak{fn}(\overline{e})$, it simply amounts to a recursive call. When the complex expression is of the form $\mathfrak{ag}(e)$, according to Section 2, one has first to find the suitable level of nesting for getting the group to be split into. Then, produce the list of values by evaluating e, and then compute the evaluation of $\mathfrak{ag}$ against this list of values. In environment $\mathcal{E}{=}[S_n; \ldots S_1]$, level i is a suitable candidate expressed by $\mathbb{S}_e(A(S_i), [S_{i-1}; \ldots; S_1], e)$ on Figure 6 whenever e is built upon $G = A(S_i) \cup \bigcup_{j<i} G(S_j)$ which is in turn

$$
\begin{array}{lcll}
[\![c]\!]^{\mathrm{f}}_{\mathcal{E}} & = & \mathrm{c} & \text{if } c \text{ is a value} \\
[\![a]\!]^{\mathrm{f}}_{[]} & = & \textbf{default} & \text{if } a \text{ is an attribute} \\
[\![a]\!]^{\mathrm{f}}_{(A,G,[])::\mathcal{E}} & = & [\![a]\!]^{\mathrm{f}}_{\mathcal{E}} & \\
[\![a]\!]^{\mathrm{f}}_{(A,G,t::T)::\mathcal{E}} & = & t.\mathrm{a} & \text{if } a \in \ell(t) \\
[\![a]\!]^{\mathrm{f}}_{(A,G,t::T)::\mathcal{E}} & = & [\![a]\!]^{\mathrm{f}}_{\mathcal{E}} & \text{if } a \notin \ell(t) \\
[\![\mathfrak{fn}(\overline{e})]\!]^{\mathrm{f}}_{\mathcal{E}} & = & [\![\mathfrak{fn}]\!]_f(\overline{[\![e]\!]^{\mathrm{f}}_{\mathcal{E}}}) & \\
 & & & \text{if } \mathfrak{fn} \text{ is a function,} \\
 & & \multicolumn{2}{r}{\text{and } \overline{e} \text{ is a list of simple expressions}}
\end{array}
$$

```
(* The type of evaluation environnements *)
Definition env_type := list (list attribute * group_by * list tuple).

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

Figure 5. Simple expressions' semantics.

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

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

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

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

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

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

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

```
Fixpoint (* (𝔹u(G, f)) *) is_built_upon G f :=
  match f with
    | F_Constant _ ⇒ true
    | F_Dot _ ⇒ f inS? g
    | F_Expr s l ⇒ (f ins? G) || forallb (is_built_upon G) l
  end.

Definition (* (𝕊e(la, env, f)) *) is_a_suitable_env la env f :=
  is_built_upon
     (map (fun a ⇒ F_Dot a) la ++
     flat_map (fun slc ⇒ match slc with (_, G, _) ⇒ G end) env)
     f.

Fixpoint (* (𝔽e(env, f)) *) find_eval_env env f :=
  match env with
    | nil ⇒ if is_built_upon nil f then Some nil else None
    | (la1, g1, l1) :: env' ⇒
      match find_eval_env env' f with
        | Some _ as e ⇒ e
        | None ⇒
          if is_a_suitable_env la1 env' f then Some env else None
      end
  end.

Fixpoint interp_aggterm env (ag : aggterm) :=
  match ag with
  | A_Expr ft ⇒ (* simple expression without aggregate *)
     interp_funterm env ft
  | A_fun f lag ⇒
    (* simple recursive call in order to evaluate independently the
     sub-expressions when the top symbol is a function *)
     interp_symb f (List.map (fun x ⇒ interp_aggterm env x) lag)
  | A_agg ag ft ⇒
    let env' :=
        if is_empty (att_of_funterm ft)
        then (* the expression under the aggregate is a constant *)
     Some env
        else (* find the outermost suitable level *) find_eval_env env
      ft in
    let lenv :=
        match env' with
          | None | Some nil ⇒ nil
          | Some ((la1, g1, l1) :: env'') ⇒
              (* the outermost group is split into *)
               map (fun t1 ⇒ (la1, g1, t1 :: nil) :: env'') l1
        end in
        interp_aggregate ag (List.map (fun e ⇒ interp_funterm e ft)
     lenv)
  end.
```

Figure 6. Complex (with aggregates) expressions' semantics.

$$
\begin{array}{lcl}
\llbracket f_1 \textbf{ and } f_2 \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \llbracket f_1 \rrbracket^{\mathrm{b}}_{\mathcal{E}} \wedge \llbracket f_2 \rrbracket^{\mathrm{b}}_{\mathcal{E}} \\
\llbracket f_1 \textbf{ or } f_2 \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \llbracket f_1 \rrbracket^{\mathrm{b}}_{\mathcal{E}} \vee \llbracket f_2 \rrbracket^{\mathrm{b}}_{\mathcal{E}} \\
\llbracket \textbf{not } f \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \neg \llbracket f \rrbracket^{\mathrm{b}}_{\mathcal{E}} \\
\llbracket \textbf{true} \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \top \\
\llbracket \mathfrak{pr}(\overline{e_i}) \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \llbracket \mathfrak{pr} \rrbracket_p (\overline{\llbracket e_i \rrbracket^{\mathrm{a}}_{\mathcal{E}}}) \\
\llbracket \mathfrak{pr}(\overline{e_i},\ \textbf{all}\ q) \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \top \\
\multicolumn{3}{r}{\text{iff } \llbracket \mathfrak{pr}(\overline{e_i}, t) \rrbracket^{\mathrm{b}}_{\mathcal{E}} = \top^{\dagger} \text{ for all } t \in \llbracket q \rrbracket^{\mathrm{q}}_{\mathcal{E}}} \\
\llbracket \mathfrak{pr}(\overline{e_i},\ \textbf{any}\ q) \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \top \\
\multicolumn{3}{r}{\text{iff } \llbracket \mathfrak{pr}(\overline{e_i}, t) \rrbracket^{\mathrm{b}}_{\mathcal{E}} = \top^{\dagger} \text{ for at least one } t \in \llbracket q \rrbracket^{\mathrm{q}}_{\mathcal{E}}} \\
\llbracket \overline{e_i \textbf{ as } a_i} \textbf{ in } q \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \top \\
\multicolumn{3}{r}{\text{if } (\overline{a_i = \llbracket e_i \rrbracket^{\mathrm{a}}_{\mathcal{E}}}) \text{ belongs to}^{\dagger}\ \llbracket q \rrbracket^{\mathrm{q}}_{\mathcal{E}}} \\
\llbracket \textbf{exists } q \rrbracket^{\mathrm{b}}_{\mathcal{E}} & = & \top \text{ iff } \llbracket q \rrbracket^{\mathrm{q}}_{\mathcal{E}} \text{ is not empty}
\end{array}
$$

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

[†] see paragraph for NULL's in Section 3.2.

Figure 7. Formulas' semantics.

expressed by $\mathbb{B}_u(G, e)$ on Figure 6. When e is a constant, the innermost level is chosen (here n), otherwise, the outermost suitable candidate level is chosen as expressed by $\mathbb{F}_e(\mathcal{E}, e)$. Formulas' semantics, given in Figure 7, relies on expressions' semantics. As the syntax is parametrised by a domain `dom`, similarly formulas' semantics is parametrised by the domain's evaluation. This is expressed, in the Coq development, by `Hypothesis I : env_type → dom → bagT.`, and is expanded as query interpretation, $\llbracket \_ \rrbracket^{\mathrm{q}}_{\_}$, in the formal definition.

Let's finally comment on query semantics, $\llbracket \_ \rrbracket^{\mathrm{q}}_{\_}$, given in Figure 8. For the set theoretic operators, we chose to assign them a bag semantics even if our notations do not explicitely mention `all`. If one wants to recover the usual set semantics for `sq = q`$_1$ `op q`$_2$, one has to apply dupplicate elimination thanks to `δ(sq) = select * from sq`$(a_i$ `as` $a_i)_{a_i \in \ell(sq)}$ `group by` ℓ`(sq)`. The most complex case is the `select from where group by having` one. Informally, a first step consists in evaluating the `from` and then filtering it thanks to the `where` formula.

More precisely how to check that a tuple t fullfils `where` condition w in context $\mathcal{E}$? According to the definition in Figure 7, w is evaluted $w.r.t$, a single environment. This means that t and $\mathcal{E}$ have to be combined into this single environment, $\mathcal{E}'$ such that $\llbracket w \rrbracket^{\mathrm{f}}_{\mathcal{E}'}$ is equal to the evaluation of w, where the attributes a in $\ell(t)$ are bound to $t.a$, and the attributes a in $\bigcup_{S \in \mathcal{E}} A(S)$ are bounded thanks to $\bigcup_{S \in \mathcal{E}} A(T)$. This is exactly what is done when $\mathcal{E}' = (\ell(t), [], [t]) :: \mathcal{E}$.

Then the (intermediate) collection of tuples obtained is partitioned according to the grouping expressions in the `group by` G, yielding a collection of collections of tuples: the groups. When there is no grouping clause, the finest partition denoted `Group_Fine` in the Coq development is used.

The way groups are further filtered $w.r.t$, the `having` condition h follows the same pattern as `where`, except that some complex expressions may occur in h. When evaluting an expression of the form $\mathfrak{ag}(e)$ for a group T, all tuples of the group are needed; when evaluting a simple expression, any tuple of T yields the same result, T being homogeneous $w.r.t$, the grouping criterion G. Hence the proper evaluation environment for filtering the group T $w.r.t$, h in environment $\mathcal{E}$ is $(\ell(T), G, T) :: \mathcal{E}$.

Last, the `select` clause is applied yielding again a collection of tuples as a result.

About **`NULL`*'s*** At the expression level, `NULL`'s are simply handled by the fact that they behave as an absorbing element $w.r.t$, functions and are simply discarded for aggregates except for `count(*)` where they contribute as 1. In our formalisation this is expressed as constraints over $\llbracket \_ \rrbracket_a$ and $\llbracket \_ \rrbracket_f$. For formulae, we used a 3-valued logic. The evaluation of $\mathfrak{pr}(\overline{e})$ in environment $\mathcal{E}$ is equal to `unknown` iff there exists e_i in $\overline{e}$ such that $\llbracket e_i \rrbracket^{\mathrm{a}}_{\mathcal{E}} =$ `NULL`. As usual, `unknown` distributes according to well-known 3-valued logic rules. Quantifiers `all` and `any` are respectively seen as a finite conjunct and a finite disjunct in 3-valued logic. Last, $\overline{e \textbf{ as } a} \textbf{ in } q$ is evaluated as a finite disjunct of $\bigwedge \overline{e = t.a}$ where t ranges in $\llbracket q \rrbracket^{\mathrm{q}}$,

$$
\begin{array}{lcll}
[\![\mathit{tbl}]\!]^{\mathrm{q}}_{\mathcal{E}} & = & [\![\mathit{tbl}]\!]_{db} & \text{if } \mathit{tbl} \text{ is a table} \\
[\![q_1 \textbf{ union } q_2]\!]^{\mathrm{q}}_{\mathcal{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathcal{E}} \cup [\![q_2]\!]^{\mathrm{q}}_{\mathcal{E}} & \\
[\![q_1 \textbf{ intersect } q_2]\!]^{\mathrm{q}}_{\mathcal{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathcal{E}} \cap [\![q_2]\!]^{\mathrm{q}}_{\mathcal{E}} & \\
[\![q_1 \textbf{ except } q_2]\!]^{\mathrm{q}}_{\mathcal{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathcal{E}} \setminus [\![q_2]\!]^{\mathrm{q}}_{\mathcal{E}} & \\
\multicolumn{4}{l}{[\![\textbf{select } \overline{e_i \textbf{ as } a_i} \textbf{ from } \overline{f_i} \textbf{ where } w \textbf{ group by } G \textbf{ having } h]\!]^{\mathrm{q}}_{\mathcal{E}} =} \\
\multicolumn{4}{c}{\left\{\!\!\left| \overline{\left(a_i = [\![e_i]\!]^{\mathrm{a}}_{(\ell(T),G,T)::\mathcal{E}} \right)} \;\middle|\; T \in \mathbb{F}_3 \right|\!\!\right\}} \\
\multicolumn{4}{l}{\quad \text{if } F = \bowtie_i [\![f_i]\!]^{\mathrm{from}}_{\mathcal{E}}} \\
\multicolumn{4}{l}{\quad \text{and } F_1 = \left\{\!\!\left| t \in F \;\middle|\; [\![w]\!]^{\mathrm{b}}_{(\ell(t),[],[t])::\mathcal{E}} = \top \right|\!\!\right\}} \\
\multicolumn{4}{l}{\quad \text{and } \mathbb{F}_2 \text{ is a partition}^{\dagger} \text{ of } F_1 \text{ according to } G} \\
\multicolumn{4}{l}{\quad \text{and } \mathbb{F}_3 = \left\{\!\!\left| T \in \mathbb{F}_2 \;\middle|\; [\![h]\!]^{\mathrm{b}}_{(\ell(T),G,T)::\mathcal{E}} = \top \right|\!\!\right\}} \\
[\![q\overline{(a_i \textbf{ as } b_i)}]\!]^{\mathrm{from}}_{\mathcal{E}} & = & \{\!\!| \overline{(b_i = c_i)} \mid \overline{(a_i = c_i)} \in [\![q]\!]^{\mathrm{q}}_{\mathcal{E}} |\!\!\} &
\end{array}
$$

[†] see paragraph for NULL's in Section 3.2.

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

(** evaluation of the from part *)
with eval_sql_from_item env x :=
  match x with
  | From_Item sqj sj ⇒
    Febag.map BTupleT BTupleT
    (fun t ⇒
       projection (env_t env t) (att_renaming_item_to_from_item sj))
    (eval_sql_query env sqj)
    end.
```

Figure 8. SQL queries' semantics.

meaning that as soon as e or $t.a$ is evaluated to null, $\bigwedge \overline{e = t.a}$ is unknown. Eventually, when used into queries' evaluation, the evaluation of formulae yielding unknown results are casted into false. It should be noticed that even if NULL is not equal to nor different from NULL or any other value in the contex of formulas, NULL is equal to NULL for grouping. This is taken into account in Figure 8 by a careful definition of partition and of make_groups in the Coq development.

3.3 Experimental assessment

Now that we have defined a formal, Coq mechanised semantics for SQL, how can we be convinced that it is correct? Rather than relying on well-known benchmarks like TPC-H [11] that mainly address performance issues whereas we seek for semantic related aspects, we adopted the approach presented in [16]. We designed an automatic query generator. The parameters of our generator are:

1. The number of tables in the schema
2. The number of attributes for each table
3. The proportion of constants among expressions
4. The max number of expressions in the select
5. The max number of queries in the from clause
6. The max number of grouping expressions in the group by
7. The max level of nesting
8. The max size of a relation's instance
9. The max integer value in in relations' instances
10. The proportion of NULL's in instances

As we needed to generate queries that were to be accepted by Postgresql and Oracle™, we not only relied on $\mathrm{SQL}_{\mathrm{Coq}}$'s grammar but also imposed well-formed conditions to our generator. We ran[9] our experiment on 10.000 queries. In all cases we observed the same results for Postgresql (Version 9.5.12), Oracle™(https://livesql.oracle.com/apex/livesql/file/index.html running Oracle Database 18c) and $\mathrm{SQL}_{\mathrm{Coq}}$. At that point we

[9] For performance reasons, as Coq is not designed for intensive computations, we used the OCaml extracted code from $\mathrm{SQL}_{\mathrm{Coq}}$.

strongly believe that $\mathrm{SQL_{Coq}}$ faithfully reflects SQL's semantics.

4 $\mathrm{SQL_{Alg}}$: a Coq mechanised algebra for SQL

We now present $\mathrm{SQL_{Alg}}$, our Coq formalisation of an algebra that hosts $\mathrm{SQL_{Coq}}$. $\mathrm{SQL_{Alg}}$ borrows from the extended relational algebra presented in [12] which consists of the well-known relational operators π (projection) which corresponds to `select`, σ (selection) corresponding to `where` and $\bowtie$ (join) to `from` together with the set theoretic operators. To account for SQL, the algebra in [12] is extended with the γ (grouping) operator.

4.1 Syntax and semantics

However, as it is presented the algebra in [12] does not account for `having` conditions neither for complex expressions (grouping is only possible over attributes and aggregates are computed over single attributes) nor for environments. Unlike this proposal, ours is far much expressive as it allows for grouping over simple expressions and allows complex expressions e^a in projections.

So as to deal with `having` conditions, that directly operate on groups that carry more information than single tuples, $\mathrm{SQL_{Alg}}$ extends what is presented in [12] by adding an extra parameter to γ: the `having` condition. As pointed in [12], one should notice that the δ (duplicate elimination) operator is absent as it is a special case of γ[10].

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

Figure 9. $\mathrm{SQL_{Alg}}$ syntax

Expressions (simple and complex ones) as well as formulas[11] are shared with $\mathrm{SQL_{Coq}}$. In order to define the semantics of $\mathrm{SQL_{Alg}}$ expressions, environments are needed, for the same reasons as for $\mathrm{SQL_{Coq}}$: accounting for nesting. $\mathrm{SQL_{Alg}}$ environments are exactly the same as for $\mathrm{SQL_{Coq}}$. What should be noticed is that $\bowtie$ is the true natural join, and that γ can be seen as a degenerated case of `select from group where by having`, where the `where` condition is absent (or set to `true`).

Let us at that point formally relate $\mathrm{SQL_{Coq}}$ and $\mathrm{SQL_{Alg}}$.

$$
\begin{array}{l}
[\![\mathit{tbl}]\!]^{\mathrm{Q}}_{\mathcal{E}} = [\![\mathit{tbl}]\!]_{db} \qquad\qquad \text{if } \mathit{tbl} \text{ is a table} \\
[\![q_1 \textbf{ union } q_2]\!]^{\mathrm{Q}}_{\mathcal{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathcal{E}} \cup [\![q_2]\!]^{\mathrm{Q}}_{\mathcal{E}} \\
[\![q_1 \textbf{ intersect } q_2]\!]^{\mathrm{Q}}_{\mathcal{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathcal{E}} \cap [\![q_2]\!]^{\mathrm{Q}}_{\mathcal{E}} \\
[\![q_1 \textbf{ except } q_2]\!]^{\mathrm{Q}}_{\mathcal{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathcal{E}} \setminus [\![q_2]\!]^{\mathrm{Q}}_{\mathcal{E}} \\
[\![q_1 \bowtie q_2]\!]^{\mathrm{Q}}_{\mathcal{E}} = \\
\left\{\!\!\left| \left(\overline{a_i = c_i}, \overline{b_j = d_j}\right) \;\middle|\; \begin{array}{l} (\overline{a_i = c_i}) \in [\![q_1]\!]^{\mathrm{Q}}_{\mathcal{E}} \wedge \\ (\overline{b_j = d_j}) \in [\![q_2]\!]^{\mathrm{Q}}_{\mathcal{E}} \wedge \\ (\forall\ i, j,\ a_i = b_j \implies c_i = d_j) \end{array} \right|\!\!\right\} \\
[\![\pi_{(\overline{e_i \textbf{ as } a_i})}(q)]\!]^{\mathrm{Q}}_{\mathcal{E}} = \{\!|\overline{(a_i = [\![e_i]\!]^{\mathrm{a}}_{(\ell(t),[],[t])::\mathcal{E}}}) \mid t \in [\![q]\!]^{\mathrm{q}}_{\mathcal{E}} |\!\} \\
[\![\sigma_f(q)]\!]^{\mathrm{Q}}_{\mathcal{E}} = \{\!| t \in [\![q]\!]^{\mathrm{q}}_{\mathcal{E}} \mid [\![f]\!]^{\mathrm{b}}_{(\ell(t),[],[t])::\mathcal{E}} = \top |\!\} \\
[\![\gamma_{(\overline{e_j \textbf{ as } a_j}, \overline{e_i}, f)}(q)]\!]^{\mathrm{q}}_{\mathcal{E}} = \\
\qquad \left\{\!\!\left| \overline{\left(a_j = [\![e_j]\!]^{\mathrm{a}}_{(\ell(T),\overline{e_i},T)::\mathcal{E}}\right)} \;\middle|\; T \in \mathbb{F}_3 \right|\!\!\right\} \\
\qquad \text{and } \mathbb{F}_2 \text{ is a partition of } [\![q]\!]^{\mathrm{Q}}_{\mathcal{E}} \text{ according to } \overline{e_i} \\
\qquad \text{and } \mathbb{F}_3 = \left\{\!\left| T \in \mathbb{F}_2 \;\middle|\; [\![f]\!]^{\mathrm{b}}_{(\ell(T),\overline{e_i},T)::\mathcal{E}} = \top \right|\!\right\}
\end{array}
$$

Figure 10. $\mathrm{SQL_{Alg}}$ semantics

4.2 $\mathrm{SQL_{Coq}}$ and $\mathrm{SQL_{Alg}}$ are equivalent

On Figure 11, we give $\mathbb{T}^{\mathrm{q}}(\_)$ a translation from $\mathrm{SQL_{Coq}}$ to $\mathrm{SQL_{Alg}}$, and its back translation $\mathbb{T}^{\mathrm{Q}}(\_)$. Both use auxilliary translations ($\mathbb{T}^{\mathrm{f}}(\_)$, resp. $\mathbb{T}^{\mathrm{F}}(\_)$) which simply traverse formulas in order to translate the queries they contain. Since simple and complex expressions are shared, they are left unchanged by these translations.

These translations are sound, provided that they are applied on "reasonable" database instances and queries.

Definition 4.1. A database instance $[\![\_]\!]_{db}$ is *well-sorted* if and only if all tuples in the same table have the same labels:
$\forall r, t_1, t_2,\ t_1 \in [\![r]\!]_{db} \wedge t_2 \in [\![r]\!]_{db} \implies \ell(t_1) = \ell(t_2)$.

Definition 4.2. A $\mathrm{SQL_{Coq}}$ query sq is *well-formed* if and only of all labels in its `from` clauses are pairwise disjoint and its sub-queries are well-formed:

$$\frac{}{\mathbb{W}^{\mathrm{q}}(\mathit{tbl})} \text{ if } \mathit{tbl} \text{ is a table}$$

$$\frac{\mathbb{W}^{\mathrm{q}}(q_1) \quad \mathbb{W}^{\mathrm{q}}(q_2)}{\mathbb{W}^{\mathrm{q}}(q_1 \textbf{ union } q_2)} \quad \frac{\mathbb{W}^{\mathrm{q}}(q_1) \quad \mathbb{W}^{\mathrm{q}}(q_2)}{\mathbb{W}^{\mathrm{q}}(q_1 \textbf{ intersect } q_2)} \quad \frac{\mathbb{W}^{\mathrm{q}}(q_1) \quad \mathbb{W}^{\mathrm{q}}(q_2)}{\mathbb{W}^{\mathrm{q}}(q_1 \textbf{ except } q_2)}$$

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

$$\frac{\mathbb{W}^{\mathrm{f}}(f_1) \quad \mathbb{W}^{\mathrm{f}}(f_2)}{\mathbb{W}^{\mathrm{f}}(f_1 \textbf{ and } f_2)} \quad \frac{\mathbb{W}^{\mathrm{f}}(f_1) \quad \mathbb{W}^{\mathrm{f}}(f_2)}{\mathbb{W}^{\mathrm{f}}(f_1 \textbf{ or } f_2)} \quad \frac{\mathbb{W}^{\mathrm{f}}(f)}{\mathbb{W}^{\mathrm{f}}(\textbf{ not } f)}$$

$$\frac{}{\mathbb{W}^{\mathrm{f}}(\textbf{true})} \quad \frac{}{\mathbb{W}^{\mathrm{f}}(\mathfrak{pr}(\overline{e}_i))} \quad \frac{\mathbb{W}^{\mathrm{q}}(q)}{\mathbb{W}^{\mathrm{f}}(\textbf{exists } q)}$$

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

[10] $\delta(Q) = \gamma_{(\overline{a \textbf{ as } a}, \overline{a}, \textbf{true})}(Q)$ where $\overline{a}$ spans over the labels of query Q.

[11] For algebraic formulas, the domain parameter `dom` is actually algebraic queries.

$$\begin{array}{l}
\mathbb{T}^{\mathrm{q}}(tbl) = tbl \\
\mathbb{T}^{\mathrm{q}}(q_1 \textbf{ union } q2) = \mathbb{T}^{\mathrm{q}}(q_1) \textbf{ union } \mathbb{T}^{\mathrm{q}}(q_2) \\
\mathbb{T}^{\mathrm{q}}(q_1 \textbf{ intersect } q2) = \mathbb{T}^{\mathrm{q}}(q_1) \textbf{ intersect } \mathbb{T}^{\mathrm{q}}(q_2) \\
\mathbb{T}^{\mathrm{q}}(q_1 \textbf{ except } q2) = \mathbb{T}^{\mathrm{q}}(q_1) \textbf{ except } \mathbb{T}^{\mathrm{q}}(q_2) \\
\mathbb{T}^{\mathrm{q}}(\textbf{select } \overline{e_i \textbf{ as } a_i} \textbf{ from } \overline{f_i} \textbf{ where } w) = \\
\qquad \pi_{(\overline{e_i \textbf{ as } a_i})}(\sigma_{\mathbb{T}^{\mathrm{f}}(w)}(\bowtie_i \overline{\mathbb{T}^{\mathrm{from}}(f_i)})) \\
\mathbb{T}^{\mathrm{q}}(\textbf{select } \overline{e_i \textbf{ as } a_i} \textbf{ from } \overline{f_i} \textbf{ where } w \\
\qquad\qquad\qquad \textbf{group by } G \textbf{ having } h) = \\
\qquad \gamma_{(\overline{e_i \textbf{ as } a_i}, G, \mathbb{T}^{\mathrm{f}}(h))}(\sigma_{\mathbb{T}^{\mathrm{f}}(w)}(\bowtie_i \overline{\mathbb{T}^{\mathrm{from}}(f_i)})) \\
\\
\mathbb{T}^{\mathrm{from}}(q\overline{(a_i \textbf{ as } b_i)}) = \pi_{\overline{(a_i \textbf{ as } b_i)}}(\mathbb{T}^{\mathrm{q}}(q))
\end{array}$$

$$\begin{array}{l}
\mathbb{T}^{\mathrm{Q}}(tbl) = tbl \\
\mathbb{T}^{\mathrm{Q}}(q_1 \textbf{ union } q2) = \mathbb{T}^{\mathrm{Q}}(q_1) \textbf{ union } \mathbb{T}^{\mathrm{Q}}(q_2) \\
\mathbb{T}^{\mathrm{Q}}(q_1 \textbf{ intersect } q2) = \mathbb{T}^{\mathrm{Q}}(q_1) \textbf{ intersect } \mathbb{T}^{\mathrm{Q}}(q_2) \\
\mathbb{T}^{\mathrm{Q}}(q_1 \textbf{ except } q2) = \mathbb{T}^{\mathrm{Q}}(q_1) \textbf{ except } \mathbb{T}^{\mathrm{Q}}(q_2) \\
\mathbb{T}^{\mathrm{Q}}(q_1 \bowtie q_2) = \\
\qquad \textbf{select } (\overline{a'_1 \textbf{ as } a_1}_{a_1 \in \ell(q_1)}, \overline{a'_2 \textbf{ as } a_2}_{a_2 \in \ell(q_2) \setminus \ell(q_1)}) \\
\qquad \textbf{from } [\mathbb{T}^{\mathrm{Q}}(q_1)(\overline{a_1 \textbf{ as } a'_1}); \mathbb{T}^{\mathrm{Q}}(q_2)(\overline{a_2 \textbf{ as } a'_2})] \\
\qquad \textbf{where } \overline{(a'_1 = a'_2)}_{a_1 \in \ell(q_1), a_2 \in \ell(q_2), a_1 = a_2} \\
\qquad\qquad\qquad \text{where } \overline{a'_1} \text{ and } \overline{a'_2} \text{ are fresh names} \\
\mathbb{T}^{\mathrm{Q}}(\pi_{(\overline{e \textbf{ as } a})}(q)) = \textbf{select } (\overline{e \textbf{ as } a}) \textbf{ from } [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \textbf{ as } a})] \\
\mathbb{T}^{\mathrm{Q}}(\sigma_f(q)) = \textbf{select } * \textbf{ from } [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \textbf{ as } a})] \textbf{where } \mathbb{T}^{\mathrm{F}}(f) \\
\mathbb{T}^{\mathrm{Q}}(\gamma_{(\overline{e \textbf{ as } a}, G, f)}(q)) = \\
\textbf{select } (\overline{e \textbf{ as } a}) \textbf{ from } [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \textbf{ as } a})] \textbf{ group by } G \textbf{ having } \mathbb{T}^{\mathrm{F}}(f)
\end{array}$$

Figure 11. Translations between $\mathrm{SQL}_{\mathrm{Coq}}$ and $\mathrm{SQL}_{\mathrm{Alg}}$.

Provided that those conditions be fullfilled we can state that the following equivalence Theorem.

Theorem 4.3. *Let* $[\![\_]\!]_{db}$ *be a* well-sorted *database instance and sq be a SQL_{Coq} query, aq a SQL_{Alg} query then:*

$$\forall \mathcal{E}, sq, \mathbb{W}^{q}(sq) \implies [\![\mathbb{T}^{q}(sq)]\!]^{Q}_{\mathcal{E}} = [\![sq]\!]^{q}_{\mathcal{E}}$$

$$\forall \mathcal{E}, aq, \ [\![\mathbb{T}^{Q}(aq)]\!]^{q}_{\mathcal{E}} = [\![aq]\!]^{Q}_{\mathcal{E}}$$

The proof proceeds by (mutual) structural induction over queries and formulas. Actually the proof is made by induction over the sizes of queries and formulas. It consists of 500 lines of Coq code and heavily rely on a tactic which allows to automate the proofs that size for sub-objects is decreasing. For the correctness of $\mathbb{T}^{\mathrm{q}}(\_)$, well-formedness hypothesis of the theorem essentially ensures that Cartesian product and natural join coincide. What was interesting is that the well-formedness hypothesis was mandatory and this shed light on the fact that, indeed, SQL `from` behaves as a cross product. For both translations, well-sortedness ensures that reasoning over tuples' labels in the evaluation of a query can be made globally, by "statically" computing the labels over a query.

5 Conclusions

Seeking for a formal semantics for SQL has been a longstanding quest for the database community. In this article, we presented a formal, Coq *mechanised*, *executable* semantics for a large realistic fragment of SQL. Both aspects, mechanised and executable, are of paramount importance.

Mechanisation inside a proof assistant, such as Coq, raises some relevant issues and enlightening questions. Indeed, as for theoretical reasons Coq requires (recursive) functions to be total, once the syntax is fixed, the semantics has to be "totally" defined. This implies that no details can be swept under the carpet. All cases must be considered. This led us to not only discover a bunch of weird queries, the ones of Figure 1, but also to discover strange boundary conditions. For instance, provided `empty` be the empty relation, query `select 1 from empty group by 1+1 having 2=2;` returns empty on both Postgresql and Oracle™. However, query `select 1 from empty having 2=2;` yields 1 in Postgresql while Oracle™ answers empty set. In that case, Postgresql consider that the partition of the emptyset $\emptyset$ is $\{\emptyset\}$ whereas Oracle™ uses the true mathematical definition: $\emptyset$. We implemented Oracle™'s semantics for this case.

Providing an *executable* semantics allows one to be convinced that the semantics is correct as it can be confronted to real systems. This is what we achieved thanks to our query generator.

Last *combining mechanisation and execution* in the same framework, namely Coq, provides the strongest possible guarantees that there are no gaps between the definition and the execution: no transcription error between a pen and paper definition and the corresponding program may occur.

Thanks to our formal semantics we have been able to relate $\mathrm{SQL}_{\mathrm{Coq}}$ and $\mathrm{SQL}_{\mathrm{Alg}}$ establishing, the first, to our best knowledge, equivalence result for that SQL fragment. Moreover, by doing so, we can recover the well-known algebraic equivalences presented in textbooks. Such equivalences are proven, using Coq, in [3].

In an early version of the development, we defined a pure set-theoretic semantics and only addressed the SQL's fragment with no duplicates. Then we addressed the bag aspects of SQL and were pleasantly surprised to discover that it was not so dramatic. Therefore, the widespread belief that *the* problem for SQL is to assign it a bag semantics is not as crucial as it seemed to be. What was really challenging was to accurately and faithfully grasp SQL's management of expressions and environments in the presence of nested queries. The ISO/IEC document was of little help along this path. On the contrary, Coq was an enlightening, very demanding master of invaluable help. Even if we knew it, it confirmed us that, SQL having initially been designed as a domain specific language intended *not* to be Turing-complete, the fact of adding more features along the time in the standardisation process, seriously, and sadly, departed it from its original elegant foundations. By formally relating SQL and an extended relational algebra, we, humbly, wanted to pay tribute to the pioneers that designed the foundational aspects of RDBMs.

Our long term goal is to provide a Coq verified SQL's compiler. The work presented in this article allows to obtain a mechanised semantic analyser that we plan to extend to features like `order by`. In [6] we provided a certification of the physical layer of a SQL engine where mainstream physical operators such as `sequential scans`, `nested loop joins`, `index joins` or `bitmap index joins` are formally specified and implemented. What remains to be done is to address the logical optimisation part of the compiler.

References

[1] T. Arvin. 2017. *Comparison of different SQL's implementations*. http://troels.arvin.dk/db/rdbms

[2] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. 2017. Handling Environments in a Nested Relational Algebra with Combinators and an Implementation in a Verified Query Compiler. In *SIGMOD Conference, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu (Eds.). ACM, 1555–1569.

[3] Véronique Benzaken and Evelyne Contejean. 2016. SQL-Cert: Coq mechanisation of SQL's compilation: Formally reconciling SQL and (relational) algebra. (Oct. 2016). https://hal.archives-ouvertes.fr/hal-01487062

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

[5] V. Benzaken, É. Contejean, and S. Dumbrava. 2017. Certifying Standard and Stratified Datalog Inference Engines in SSReflect. In *8th International Conference on Interactive Theorem Proving (ITP 2017)*, M. Ayala-Rincon and C. Munoz (Eds.), Vol. 10499. Springer.

[6] V. Benzaken, É. Contejean, C. Keller, and E. Martins. 2018. A Coq formalisation of SQL's execution engines. In *International Conference on Interactive Theorem Proving (ITP 2018)*. Oxford, United Kingdom.

[7] S. Ceri and G. Gotlob. 1985. Translating SQL into Relational Algebra: Optimisation, Semantics, and Equivalence of SQL Queries. *IEEE Trans., on Software Engineering* SE-11 (April 1985), 324–345.

[8] J. Cheney and Ch. Urban. 2011. Mechanizing the Metatheory of mini-XQuery. In *CPP*. 280–295.

[9] S. Chu, K. Weitz, A. Cheung, and D. Suciu. 2017. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. In *PLDI 2017*. ACM, New York, NY, USA, 510–524.

[10] S. Cluet and G. Moerkotte. 1993. Nested Queries in Object Bases. In *Database Programming Languages (DBPL-4), Manhattan, New York City, USA, 30 August - 1 September 1993*. 226–242.

[11] Transaction Processing Performance Concil. [n. d.]. *TPC Benchmark (Decision Support) Standard Specification Revision 2.17*. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf

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

[13] C. Gonzalia. 2003. Towards a Formalisation of Relational Database Theory in Constructive Type Theory. In *RelMiCS (LNCS)*, R. Berghammer, B. Möller, and G. Struth (Eds.), Vol. 3051. Springer, 137–148.

[14] C. Gonzalia. 2006. *Relations in Dependent Type Theory*. Ph.D. Dissertation. Chalmers Göteborg University.

[15] T J. Green, G. Karvounarakis, and V. Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*. 31–40.

[16] P. Guagliardo and L. Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *PVLDB* 11, 1 (2017), 27–39.

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

[18] ISO/IEC. 2006. Information technology - Database Languages - SQL - Part 2: Foundation (SQL/Foundation). Final Commitee Draft.

[19] H. Katz, D. Chamberlin, M. Kay, Ph. Wadler, and D. Draper. 2003. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[20] X. Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446.

[21] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. 2010. Toward a Verified Relational Database Management System. In *ACM Int. Conf. POPL*.

[22] M. Negri, G. Pelagatti, and L. Sbattella. 1991. Formal Semantics of SQL Queries. *ACM Trans. Database Syst.* 16, 3 (1991), 513–534.

[23] B. Pierce et al. 2018. *Software Foundations - Programming Languages Foundations*. Vol. 2.

[24] The Agda Development Team. 2010. *The Agda Proof Assistant Reference Manual*. http://wiki.portal.chalmers.se/agda/pmwiki.php

[25] The Coq Development Team. 2010. *The Coq Proof Assistant Reference Manual*. http://coq.inria.fr

[26] The Isabelle Development Team. 2010. *The Isabelle Interactive Theorem Prover*. https://isabelle.in.tum.de/

[27] X. Yang, Y. Chen, E. Eide, and J. Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294.