Translating canonical SQL to imperative code in Coq

Véronique Benzaken, Évelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, Jérôme Siméon

Translating Canonical SQL to Imperative Code in Coq

VÉRONIQUE BENZAKEN, LMF, Université Paris-Saclay, France
ÉVELYNE CONTEJEAN, LMF, CNRS, Université Paris-Saclay, France
MOHAMMED HOUSSEM HACHMAOUI, LMF, Université Paris-Saclay, France
CHANTAL KELLER, LMF, Université Paris-Saclay, France
LOUIS MANDEL, IBM Research, USA
AVRAHAM SHINNAR, IBM Research, USA
JÉRÔME SIMÉON*, DocuSign, Inc., USA

SQL is by far the most widely used and implemented query language. Yet, on some key features, such as correlated queries and `NULL` value semantics, many implementations diverge or contain bugs. We leverage recent advances in the formalization of SQL and query compilers to develop DBCert, the first mechanically verified compiler from SQL queries written in a canonical form to imperative code. Building DBCert required several new contributions which are described in this paper. First, we specify and mechanize a complete translation from SQL to the Nested Relational Algebra which can be used for query optimization. Second, we define Imp, a small imperative language sufficient to express SQL and which can target several execution languages including JavaScript. Finally, we develop a mechanized translation from the nested relational algebra to Imp, using the nested relational calculus as an intermediate step.

CCS Concepts: • **Software and its engineering → Semantics**; **Compilers**; **Formal software verification**; • **Information systems → Structured Query Language**.

Additional Key Words and Phrases: Semantics preserving compiler, Query compiler, SQL, JavaScript, Coq

ACM Reference Format:
Véronique Benzaken, Évelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2022. Translating Canonical SQL to Imperative Code in Coq. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 83 (April 2022), 27 pages. https://doi.org/10.1145/3527327

1 INTRODUCTION

SQL is by far the most widely used query language. While originally designed to query relational databases, it is now also used for data integration [Lee et al. 2016], for processing logs [Jin-De 2010], and for big data [Grover et al. 2015]. It is commonly available as a library in a number of programming languages [AlaSQL 2022; SQLAlchemy 2021].

While the SQL semantics for flat select-project-join queries are well understood and consistent across platforms, several important features such as nested queries (often called correlated queries in the database literature) and `NULL` values are a common source of bugs [Benzaken and Contejean

*This author's work conducted while at Clause, Inc.

Authors' addresses: Véronique Benzaken, LMF, Université Paris-Saclay, France, veronique.benzaken@universite-paris-saclay.fr; Évelyne Contejean, LMF, CNRS, Université Paris-Saclay, France, Evelyne.Contejean@lri.fr; Mohammed Houssem Hachmaoui, LMF, Université Paris-Saclay, France, mohammed.hachmaoui@lri.fr; Chantal Keller, LMF, Université Paris-Saclay, France, Chantal.Keller@lri.fr; Louis Mandel, IBM Research, USA, lmandel@us.ibm.com; Avraham Shinnar, IBM Research, USA, shinnar@us.ibm.com; Jérôme Siméon, DocuSign, Inc., USA, jerome.simeon@docusign.com.

2475-1421/2022/4-ART83
https://doi.org/10.1145/3527327

2019; Guagliardo and Libkin 2017]. Since SQL is commonly used in critical applications and for handling sensitive data, like accessing medical information, SQL implementations can benefit from the use of formal verification techniques.

Despite recent progress in mechanized semantics for query languages in general [Shinnar et al. 2015] and for SQL in particular [Benzaken and Contejean 2019; Chu et al. 2017], those are far from being usable as a SQL implementation, typically missing a query optimizer and the ability to generate efficient code. In this paper, we describe DBCert, a compiler from SQL to imperative code which addresses those limitations and is mechanically verified using the Coq proof assistant.

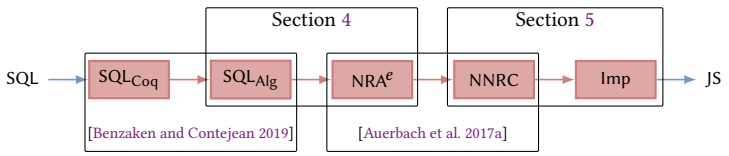
To build DBCert, we followed a classical database compiler architecture [Shaikhha et al. 2016]: (1) a source language to write the queries, (2) an algebra suitable for optimization, and (3) a physical plan to execute the queries, specific to the targeted runtime. Our compiler enhances this architecture by formally defining each of these components and proving the translation between them correct.

For the source language (1), we use the mechanized SQL semantics from Benzaken and Contejean [2019]. To our knowledge, this is the most complete formal semantics of SQL currently available. That semantics is fully executable and supports a large subset of SQL, including: `select from where group by having` blocks, `NULL` values, aggregate functions, and correlated queries.[1]

For the intermediate algebra (2), we use the Nested Relational Algebra (NRA^e) from Auerbach et al. [2017a] which comes with a Coq mechanization, including a multi-step, optimizing compiler. NRA^e has several features essential to capture SQL queries. First it can naturally handle nested queries. Second, it includes operators over sum types which we use to encode the semantics of `NULL` values. Finally, NRA^e was chosen for its validated use for optimization of nested queries [Claußen et al. 1997; Cluet and Moerkotte 1993; Moerkotte 2020] and `NULL` values [Claußen et al. 2000].

For the physical plan (3), we generate JavaScript code to target an in-memory database where the data are represented as JSON objects. We chose JavaScript for our final output because it is very portable. For the formalization, we defined a series of intermediate languages that progressively change to programming model from NRA^e to a small imperative language Imp, which is sufficiently expressive to capture SQL semantics. Imp is parameterized by a data model and a set of operators which makes it flexible enough to drive code generation for a range of target execution languages.

Architecture. The following diagram outlines the full DBCert compilation pipeline.



The compilation from SQL_{Coq}, a canonical form for SQL queries, to Imp is fully verified: the semantics of any valid SQL_{Coq} query is preserved by the compilation. SQL_{Coq} is first compiled to SQL_{Alg}, an extension of the relational algebra that includes a SQL grouping operator, formulas and environment handling. That algebra is translated to NRA^e, which is used for query optimization. Finally, NRA^e is translated to the small imperative language Imp. This translation is decomposed into multiple steps using different intermediate languages, starting with the Named Nested Relational Calculus (NNRC) [Van den Bussche and Vansummeren 2007], a functional language with list comprehensions. Using NNRC is a pragmatic choice, as it is proven equivalent to NRA^e [Auerbach et al. 2017a] and is closer to traditional languages, having variable names instead of just using combinators.

[1]The constructs that are not handled yet which represent a loss of expressiveness are: silent coercion from singleton bags to values, recursive queries, `distinct`, and `order by`.

To make the resulting compiler usable, it is complemented by non-verified front- and back-ends: a parser for SQL into SQL_{Coq}, which also performs simple disambiguation (*e.g.* to avoid name clashes) and a code generation step from Imp to JavaScript. The generated code by the compiler linked to a runtime can then be used as a Node.js library.

Contributions. This paper makes the following contributions:

- A translation from SQL to the optimizing algebra NRA^e. To the best of our knowledge, there is no description (including in database literature) of such a translation for such a large fragment of SQL, and we will see that this step is not trivial. This part represents about 20,000 lines of new formalization and proofs.
- A translation from NRA^e to a physical plan for an in-memory database. This includes the definition of multiple intermediate languages, introduced to break the difficulty of the proofs, and the language Imp, a simple imperative language sufficiently expressive to capture SQL semantics. In addition to the resulting correct-by-construction back-end, this is an advance in verification techniques. This part represents around 31,000 lines of new formalization and proofs.
- A complete compiler for a large subset of SQL, written in Coq, that translates to a database algebra suitable for optimization, and generates low-level imperative code for execution. This bridges a gap between prior works on SQL formalization and on mechanization of query compilers [Auerbach et al. 2017b; Malecha et al. 2010].

While we rely heavily on prior work, building DBCert required significant new development. First, we had to bridge the gap between the source SQL semantics and the algebraic intermediate representation in NRA^e. In particular, NRA^e does not have a builtin for `NULL` and must have an explicit encoding of the SQL environments semantics. Second, we had to develop translations and correctness proofs from the algebraic representation to a lower-level imperative language. This proof necessitated the creation of a series of intermediate languages to cope with its complexity.

Outline. Section 2 presents some simple SQL examples illustrating subtleties of the semantics of the language (Section 2.1) and then explains the compilation pipeline on an example (Section 2.2). Section 3 defines the main languages used in the compiler. Section 4 describes the translation from SQL_{Alg} to NRA^e, This translation handles delicate aspects of SQL, including `NULL` values and environments for correlated queries. Section 5 describes the translation from NNRC, an expression oriented functional language, to Imp, a statement oriented imperative language with mutable variables. Section 6 reviews the DBCert implementation. The compiler is verified using the Coq proof assistant and extracted to OCaml. The non-verified parts include the SQL parser, written in OCaml, and the JavaScript code generation and runtime used for execution. Section 7 evaluates the compiler on some challenging queries and discusses methodology. This paper provides insight both on the compilation of SQL, and on the software engineering aspect of connecting two large Coq projects developed independently.

An extended version of this article with appendix is available [Benzaken et al. 2022b]. This article is also accompanied with an artifact [Benzaken et al. 2022a] which is a Docker image containing an installed version of a snapshot of the following open source projects:

- https://github.com/dbcert/dbcert: the entry point of the project containing the main theorem and providing the runtime and the runner
- https://framagit.org/formaldata/sqltonracert: the frontend from SQL to NRA^e (which itself relies on the SQL formal semantics)
- https://querycert.github.io: the Q*Cert compiler that contains in particular the backend from NRA^e to Imp.

2 OVERVIEW

2.1 Challenges of the SQL Semantics

As an introductory example, let us consider the following SQL query written using the AlaSQL [AlaSQL 2022] library for Node.js:

```
alasql('CREATE TABLE R (a number, b number)');
alasql.tables.R.data = [ {a: 1, b: 10}, {a: 2, b: 20}, {a: 3, b: 30} ];
var res = alasql('select a from R where b > 15'); // res = [ { "a": 2 }, { "a": 3 } ]
```

This library is primarily used for querying JSON data in memory or to run SQL queries directly in the browser. The first line declares a relational schema with one table `R` containing two columns `a` and `b`, both of type number. The second line populates the database, here as a JavaScript array of objects, where each object corresponds to a row in table `R` with those same fields `a` and `b`. The third line executes a simple `select from where` statement which returns the `a` column for every row which has a `b` column greater than 15.

Challenges with null value semantics. We next consider a query adapted from Guagliardo and Libkin [2017] involving `NULL` values, an important feature of SQL commonly used to model missing data. This query selects all the values in the table `R` that are for sure not in `S`.

```
alasql('CREATE TABLE R (a number)');
alasql('CREATE TABLE S (b number)');
alasql.tables.R.data = [ {a: 1}, {a: null} ];
alasql.tables.S.data = [ {b: null} ];
var res = alasql('select a from R where a not in (select b from S)');
// expected: res = []
// alasql:   res = [ {"a": 1} ]
```

The nested query `select b from S` returns a table with a single row containing `null`. For each row in `R`, the value of the attribute `a` is compared to `null` to test non-membership in the result of `select b from S`. The `not in` predicate expands to $1 \neq \texttt{null}$ for the first row and $\texttt{null} \neq \texttt{null}$ for the second row. SQL uses a three-valued logic and comparing a value to `NULL` returns *unknown*. Thus, the two comparisons return *unknown* and the query's result should be the empty collection.

Unfortunately, AlaSQL, when given this query, incorrectly returns `[{a:1}]` instead. AlaSQL probably relies on the JavaScript comparison where $1 \neq \texttt{null}$ is *true* instead of *unknown*.

Challenges with nested query semantics. A main challenge when compiling SQL to JavaScript is correctly handling nested queries. In particular, extra care needs to be taken to account for correlated queries, where the inner query refers to a variable introduced by an outer query.

Correlated queries are an important feature of SQL since they allow in particular to answer negative questions like the previous query (where the correlation is introduced by the `in` operator).

To illustrate the subtlety of the semantics of nested queries, we consider the following queries `Q1` and `Q2`. The only difference between them is in the expression `sum(1+0*b)`: the variable b refers to `b2` (defined in table `t2`) in `Q1` and to `b1` (defined in table `t1`) in `Q2`.

```
-- Q1
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*b2) = 2);
```

```
-- Q2
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*b1) = 2);
```

t1		t2		Q1	Q2
a1	b1	a2	b2	a1	a1
1	1	7	7	1	1
1	2	7	8		
2	3			2	
3	1			3	
3	2				
3	3				

At first glance, adding a term equal to 0 in a sum should have no effect. However, `sum` is an aggregate operator, and is executed on each element of the table containing b. Thus, the expression `sum(1+0*b)` effectively counts the number of occurrences of b. This expression therefore returns different results when applied to table `t1` (`sum(1+0*b1)`, as in Q2) and `t2` (`sum(1+0*b2)`, as in Q1).

The semantics of `t1 group by a1` is to split the table `t1` into intermediate tables where the values of the attribute `a1` is the same. In our example, for both queries, it creates the tables `[{a1:1, b1:1}, {a1:1, b1:2}]`, `[{a1:2, b1:3}]`, and `[{a1:3, b1:1}, {a1:3, b1:2}, {a1:3, b1:3}]`. Then, on each of these tables, the condition `having` is executed. The expression `select a2 from t2 group by a2 having sum(1+0*b) = 2` is thus executed three times in three different contexts. The expression `t2 group by a2` always creates the table `[{a2:7, b2:7}, {a2:7, b2:8}]` and the condition `having sum(1+0*b) = 2` tests if the number of occurrences of b is two.

For `Q1`, where b = `b2`, since there are three times the condition is true (since the table containing `b2` has two elements), the inner query returns `[{a2:7}]`, and the `exists` condition is always a success. As a result, the outer query returns `[{a1:1}, {a1:2}, {a1:3}]`.

For `Q2`, where b = `b1`, the table containing `b1` has two elements such that `a1 = 1`, one element such that `a1 = 2`, and three elements such that `a1 = 3`. The condition is true only once (when `a1 = 1`), so the inner query returns `[{a2:7}]` and the `exists` condition succeeds only in this case. Thus, the outer query returns `[{a1:1}]`. AlaSQL, alas, produces an incorrect result for this query.

This example illustrates what we call the *environment handling* of SQL: evaluating a nested query must be done in an environment aware of all outer queries, and one must be careful on correctly choosing the important piece of information in this environment. We will detail this in Sections 3.1 and 4.3.

2.2 Translating SQL to JavaScript

We introduce our translation using a simple correlated query that returns all the values of the attribute `a` of the table `R` which are present in the column `b` of `S`:

```
select a from R where exists (select b from S where b = a);
```

We first translate the query into SQL_{Coq}, a subset of SQL where all implicit features of SQL are explicit. For example, a `select` without a `where` clause is completed by `where true`. SQL_{Coq} is as expressive as the considered subset of SQL, but its regularity simplifies formalization. Our example in SQL_{Coq} becomes:

```
select x as a from (table R) t0(x)
  where exists (select y as t1_y from (table S) t1(y) where y = x);
```

In SQL_{Coq}, all the intermediate results must be named. For example, the notation `(table R) t0(x)` renames the table `R` into `t0` and its single attribute is renamed `x`. The selection `x as a` projects the attribute `x` of the table `t0` and renames it `a`. The definition of SQL_{Coq} and this compilation step is

taken from Benzaken and Contejean [2019]. Currently, the SQL features not supported by SQL_{Coq} are silent coercion from singleton bags to values, recursive queries, `distinct`, and `order by`.

The next compilation step is also taken from Benzaken and Contejean [2019]. It translates SQL_{Coq} to SQL_{Alg}, a relational algebra such as is found in database textbooks [Abiteboul et al. 1995; Ullman 1982], but including grouping and aggregates. SQL_{Alg} includes operators such as projection π, selection σ, natural join $\bowtie$, and a grouping operator γ. Our example query translated into SQL_{Alg} is:

$$\pi_{x \text{ as } a}(\sigma_{\texttt{exists}(\pi_{y \text{ as } t1\_y}(\sigma_{y=x}(\pi_{b \text{ as } y}(S))))}(\pi_{a \text{ as } x}(R)))$$

The input of this query is the expression $\pi_{a \text{ as } x}(R)$, corresponding to `(table R) t0(x)`, the renaming of the attribute of the table `R`. The top-level $\pi_{x \text{ as } a}$ corresponds to the projection of the result by the clause `select x as a`. It is applied to $\sigma_{\texttt{exists}(\ldots)}$, which corresponds to the `where exists (...)` clause. Similarly, the expression inside the `exists` predicate corresponds to the inner SQL_{Coq} query.

From SQL_{Alg}, the query is translated into NRA^e [Auerbach et al. 2017a], a nested relational algebra. This intermediate language has two purposes: (1) it makes explicit the encoding of SQL features like `NULL` values and the environment handling, and (2) it is a good language for optimization [Cao and Badia 2007; Moerkotte 2020]. NRA^e is based on functional combinators, evaluated in a context with exactly two variables: `In` for the current input and `Env` for the local environment. The intuition for that translation is that the structure of relational algebra operators (e.g., π, σ) is preserved, but "administrative" steps are added to deal with `NULL` values and the SQL evaluation context is encoded in the NRA^e environment `Env`.

Consider first the translation of $\pi_{a \text{ as } x}(R)$ from SQL_{Alg} to NRA^e:

$$\chi_{\langle\{x:\texttt{In}.a\}\rangle}(R)$$

The combinator χ is a functional map: it applies the expression within the $\langle..\rangle$ to each element of R where the element is bound to the variable `In`, which holds the current input. The expression $\{x : \texttt{In}.a\}$ creates a record with label x and value the projection of the label a from the current input (`In`). As expected, this expression creates a collection of records with label x containing the elements of R with label a.

We next consider the translation of $\sigma_{\texttt{exists}(Q)}(\pi_{a \text{ as } x}(R))$ where Q has S as input, but also depends on x, the result of $\pi_{a \text{ as } x}(R)$. Denoting the translation of Q as q, $\sigma_{\texttt{exists}(Q)}(\pi_{a \text{ as } x}(R))$ is:

$$\sigma_{\langle \texttt{exists}(q) \circ^e \text{push}_{one}\rangle}\left(\chi_{\langle\{x:\texttt{In}.a\}\rangle}(R)\right)$$

The selection operator σ of SQL_{Alg} is translated into the corresponding operator in NRA^e. But in SQL_{Alg}, the σ operator implicitly adds x to the evaluation context of $\texttt{exists}(q)$. This is done explicitly in NRA^e, with $q_1 \circ^e q_2$, which evaluates q_2 first and then evaluates q_1 in the environment `Env` where the result of q_2 is stored. Here, push_{one} adds the value of x onto a stack defining the evaluation context of $\texttt{exists}(q)$, implemented as a linked list with shape $\{slice : \bullet, tail : \bullet\}$. *slice* contains the attributes introduced by $\chi_{\langle\{x:\texttt{In}.a\}\rangle}(R)$ and *tail* contains the evaluation context of $\chi_{\langle\{x:\texttt{In}.a\}\rangle}(R)$.

Finally, the last difficulties are in the translation of the expression $\sigma_{y=x}(\ldots)$, which has the following structure in NRA^e (to simplify the presentation we simplified the code: in particular, we assume that y is not `NULL`):

$$\sigma_{\langle(\texttt{In}|\mathit{false}) \circ ((\texttt{left (Env}.\mathit{slice}.y=\texttt{In})|\texttt{right ()}) \circ \texttt{Env}.\mathit{tail}.\mathit{slice}.x) \circ^e \text{push}_{one}\rangle}(\ldots)$$

Starting from the right of the condition of the σ, the $\circ^e \text{push}_{one}$ adds y on the top of the environment stack. Then $(q \circ \texttt{Env}.\mathit{tail}.\mathit{slice}.x)$ where $q = (\texttt{left (Env}.\mathit{slice}.y = \texttt{In})|\texttt{right ()})$ corresponds to the equality test $y = x$ which has to deal with `NULL` values. The expression $\texttt{Env}.\mathit{tail}.\mathit{slice}.x$ accesses the value of x at the appropriate level in the environment stack and is given as input to q using

the composition operator $\circ$. The expression q tests if x is null (we assume here that y is not null). NRA^e does not have a built-in notion of null, instead values that can be `NULL` are boxed in a value of type `either`. For example, the number 42 is encoded as `left` 42 and a `NULL` is `right ()`. The operator $(q_1|q_2)$ matches its input with shape `left` d_1 and `right` d_2 to execute either q_1 with input d_1 or q_2 with input d_2. In our example, if x is null, q returns `right ()` (corresponding to *unknown*), otherwise `left` (`Env`.*slice*.y = `In`). The expression `Env`.*slice*.y = `In` performs the comparison $y = x$ knowing that x and y are not null. The output of q is of type `either`, representing the three-valued logic of SQL: `left` *true* is *true*, `left` *false* is *false*, and `right ()` is *unknown*. The expression (`In`|*false*) in the σ converts a boxed three-valued logic value to a Boolean.

From NRA^e, the translation rewrites the query through a series of intermediate languages that are successively closer to Imp, a simple imperative language. Imp contains variables, assignments, conditionals, iterations, and calls to external operators and external runtime functions. It supports compilation from SQL while remaining easy to translate into various imperative languages.

The translation of a query in Imp produces a function which is parameterized by the database instance represented as a record where each table is a field. The body of the function initializes a variable `ret` with the result of the query which is returned at the end.

```
fun(db) {
  var R = db.R; var ret; ... return ret;
}
```

In Imp, the code corresponding to $\chi_{\langle\{x:\mathtt{In}.a\}\rangle}(R)$ is a loop that builds a collection `tmp0` by iterating over the input collection provided in `R`.

```
var tmp0 = array(); for (id0 in R) { tmp0 = push(tmp0, { x : id0.a }) }
```

Finally, from Imp, we obtain JavaScript code that executes the query via a straightforward translation. This is linked to a JavaScript runtime that implements operations like `array` and `push`.

3 MAIN LANGUAGES

SQL is a declarative query language built around the famous `select from where` statement. While most formal treatments use a set-theoretic semantics, SQL implementations use a *bag* semantics, i.e., unordered collection in which the same element may occur multiple times. Most realistic queries use significantly more complex features, including `select from where group by having` statements to handle aggregation and collection operators such as $\cup$ (`union`), $\cap$ (`intersect`) and $\setminus$ (`except`) . SQL queries have also to account for `NULL` values that are used to represent unknown or missing information in tables. SQL queries involve *predicates* (`=`,`<`,...) *functions* (`+`,`-`,...) and *aggregates* (`sum`, `max`, `count`,...) which are often used in conjunction with `group by having`. Last, SQL allows for nested expressions, e.g., queries inside the `where` or `having` clause.

To compile SQL, the three languages that are at the core of the contribution are SQL$_{\text{Alg}}$, NRA^e, and Imp. We present their syntax, data models, and semantics.

3.1 SQL$_{\text{Alg}}$

SQL$_{\text{Alg}}$ [Benzaken and Contejean 2019] is an extension of the relational algebra to encompass SQL's aggregates, formulas, bag semantics, and environment handling. The goal of SQL$_{\text{Alg}}$ is to capture the semantics of SQL using the relational algebra as in database textbooks [Abiteboul et al. 1995; Ullman 1982], but on a larger fragment than is typically presented.

SQL$_{\text{Alg}}$ queries operate on a flat data model. A database instance is a set of named *relations* (or *tables*). Each table is a bag of *tuples* where each element of the tuple is a raw value (number, string,

$$[\![()]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) = \{|\ |\} \qquad [\![tbl]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) = i.tbl \quad \text{if } tbl \text{ is a table}$$

$$[\![Q_1 \bowtie Q_2]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) = \left\{\!\!\left| \left(\overline{a_n = c_n}, \overline{b_k = d_k}\right) \;\middle|\; \begin{array}{l} (\overline{a_n = c_n}) \in [\![Q_1]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) \wedge \\ (\overline{b_k = d_k}) \in [\![Q_2]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) \wedge \\ (\forall\ n, k,\ a_n = b_k \Rightarrow c_n = d_k) \end{array} \right|\!\!\right\}$$

$$[\![\pi_{(\overline{e_n \text{ as } a_n})}(Q)]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) = \{|(\overline{a_n = [\![e_n]\!]^{\mathrm{e}}_{(\ell(t),[],[t])::\mathscr{E}}}) \mid t \in [\![Q]\!]^{\mathrm{Q}}_{\mathscr{E}}(i)|\}$$

$$[\![\sigma_f(Q)]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) = \{|t \in [\![Q]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) \mid [\![f]\!]^{\mathrm{f}}_{(\ell(t),[],[t])::\mathscr{E}}(i) = \top|\}$$

$$[\![\gamma_{(\overline{e_k \text{ as } a_k}, \overline{e_n}, f)}(Q)]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) = \left\{\!\!\left| (\overline{a_k = [\![e_k]\!]^{\mathrm{e}}_{(\ell(T),\overline{e_n},T)::\mathscr{E}}}) \;\middle|\; T \in \mathbb{F}_3 \right|\!\!\right\}$$

and $\mathbb{F}_2$ is a partition of $[\![Q]\!]^{\mathrm{Q}}_{\mathscr{E}}(i)$ according to $\overline{e_n}$

and $\mathbb{F}_3 = \left\{\!\!\left| T \in \mathbb{F}_2 \;\middle|\; [\![f]\!]^{\mathrm{f}}_{(\ell(T),\overline{e_n},T)::\mathscr{E}}(i) = \top \right|\!\!\right\}$

$$[\![f_1 \text{ and } f_2]\!]^{\mathrm{f}}_{\mathscr{E}}(i) = [\![f_1]\!]^{\mathrm{f}}_{\mathscr{E}}(i) \wedge_3 [\![f_2]\!]^{\mathrm{f}}_{\mathscr{E}}(i) \qquad [\![p(\overline{e_n},\ \texttt{all}\ q)]\!]^{\mathrm{f}}_{\mathscr{E}}(i) = \textit{true} \text{ iff } [\![p(\overline{e_n}, t)]\!]^{\mathrm{f}}_{\mathscr{E}}(i) = \textit{true} \text{ for all } t \in [\![q]\!]^{\mathrm{Q}}_{\mathscr{E}}(i)$$

$$[\![f_1 \text{ or } f_2]\!]^{\mathrm{f}}_{\mathscr{E}}(i) = [\![f_1]\!]^{\mathrm{f}}_{\mathscr{E}}(i) \vee_3 [\![f_2]\!]^{\mathrm{f}}_{\mathscr{E}}(i) \qquad [\![\texttt{exists}\ q]\!]^{\mathrm{f}}_{\mathscr{E}}(i) = \textit{true} \text{ iff } [\![q]\!]^{\mathrm{Q}}_{\mathscr{E}}(i) \text{ is not empty}$$

$$[\![p(\overline{e_n})]\!]^{\mathrm{f}}_{\mathscr{E}}(i) = p(\overline{[\![e_n]\!]^{\mathrm{a}}_{\mathscr{E}}})$$

$$[\![c]\!]^{\mathrm{e}}_{\mathscr{E}} = \mathrm{c} \qquad [\![\mathfrak{fn}(\overline{e})]\!]^{\mathrm{e}}_{\mathscr{E}} = \mathfrak{fn}(\overline{[\![e]\!]^{\mathrm{e}}_{\mathscr{E}}}) \qquad [\![a]\!]^{\mathrm{e}}_{(A,G,T)::\mathscr{E}} = \begin{cases} T.a & \text{if } a \in A \\ [\![a]\!]^{\mathrm{e}}_{\mathscr{E}} & \text{if } a \notin A \end{cases}$$

Fig. 1. Semantics of $\mathrm{SQL}_{\mathrm{Alg}}$ (excerpt).

etc) and can be accessed with its name called *attribute*. All the tuples in a table have the same set of attributes, the `NULL` value is used to encode a missing attribute.

The syntax of $\mathrm{SQL}_{\mathrm{Alg}}$ is the following (the notation $\overline{e}$ indicates a list of expressions e).

$$\begin{array}{rcl} Q & ::= & () \mid tbl \\ & & \mid Q\ (\texttt{union} \mid \texttt{intersect} \mid \texttt{except})\ Q \mid Q \bowtie Q \\ & & \mid \pi_{(\overline{e\ \texttt{as}\ a})}(Q) \mid \sigma_f(Q) \\ & & \mid \gamma_{(\overline{e\ \texttt{as}\ a}, \overline{e}, f)}(Q) \end{array} \qquad \begin{array}{rcl} f & ::= & \texttt{true} \mid f\ (\texttt{and} \mid \texttt{or})\ f \mid \texttt{not}\ f \\ & & \mid p(\overline{e}) \mid p(\overline{e}, (\texttt{all} \mid \texttt{any})\ Q) \\ & & \mid \overline{e\ \texttt{as}\ a}\ \texttt{in}\ Q \mid \texttt{exists}\ Q \\ e & ::= & c \mid a \mid \mathfrak{fn}(\overline{e}) \mid \mathfrak{ag}(\overline{e}) \end{array}$$

A query Q can be a tuple with no attributes ($()$), a relation name tbl, a set operation on two sub-queries, a natural join $\bowtie$,[2] a projection (and renaming) π, a selection sigma σ, or a grouping γ. The γ operator extends the standard relational algebra with the possibility to compute groups and aggregates similarly to a `select`/`group by`/`having` in SQL. A formula f is an expression returning a Boolean value in the three-valued logic where p is a predicate (e.g., $<$). An expression e can be a constant c from the set of values $\mathscr{V}$ (currently DBCert supports intergers, Booleans, and string and also floating point numbers in a separate version), an attribute name a corresponding to a component of a tuple, or a function call. There are two classes of functions: (1) the functions $\mathfrak{fn}$ that combine values (like +, −, ∗), and (2) aggregate functions $\mathfrak{ag}$ that operate over collections (like `sum`, `avg`, or `min`).[3]

The semantics of $\mathrm{SQL}_{\mathrm{Alg}}$ is defined in Figure 1 (the full definition is given in the extended version of the paper [Benzaken et al. 2022b]). The semantics function of each syntactic category is annotated with the syntactic category of the term (Q for queries, f for formulas, and e for expressions). The semantics $[\![Q]\!]^{\mathrm{Q}}_{\mathscr{E}}(i)$ of a query Q evaluated in an environment $\mathscr{E}$ on a database instance i defines a

[2] A natural join $Q_1 \bowtie Q_2$ computes the set of all combinations of tuples in Q_1 and Q_2 that are equal on their common attribute names. For example, if Q_1 computes the bag of tuples $\{|(a:1, b:2), (a:2, b:2), (a:3, b:3)|\}$ and Q_2 computes the bag $\{|(b:1, c:1), (b:2, c:2), (b:2, c:3)|\}$, then their natural join is the bag $\{|(a:1, b:2, c:2), (a:2, b:2, c:2), (a:1, b:2, c:3), (a:2, b:2, c:3)|\}$. For instance, the tuple $(a:3, b:3)$ from Q_1 is discarded since there is no tuple in Q_2 whose value on the attribute b is 3, whereas the tuple $(a:1, b:2)$ is combined with all the tuples of Q_2 whose value on the attribute b is 2.

[3] In the implementation, expressions with and without aggregate are syntactically stratified.

bag. The instance i associates the data to each table. The environment $\mathscr{E}$ defines the local evaluation context of the query, and is a major subtlety in the semantics of SQL that we now detail.

An environment $\mathscr{E} = [S_n; ...; S_1]$ has a stack structure reflecting the current nesting level of the query. Each level of the stack is a *slice* $S = (A, G, T)$ where A is the set of attributes defined at the slice level, G is the list of grouping expressions in the case of a γ, and T is the current tuple (or list of tuples for a γ) the query is evaluated against. We use the notation $\mathscr{E} = S :: \mathscr{E}'$ to access the top of the stack and $A(S)$, $G(S)$, $T(S)$ to access to the different elements of a slice.

The rules for the projection ($\pi_{(\overline{e_n \text{ as } a_n})}(Q)$) and for the grouping operator ($\gamma_{(\overline{e_k \text{ as } a_k}, \overline{e_n}, f)}(Q)$) in Figure 1 illustrate the construction of the environment. Projection builds a bag by iterating on each tuple t in the result of the evaluation of Q. For each t, it creates a new tuple with attributes $\overline{a_n}$ of value $\overline{e_n}$. Each expression e_n is evaluated in an environment $\mathscr{E}' = (\ell(t), [], [t]) :: \mathscr{E}$ where the function ℓ extracts the attribute names of t. The grouping operator, in contrast, first agglomerates tuples as per some grouping expressions $\overline{e_n}$, then filters out some of the groups using the predicate f, and finally computes the resulting expressions $\overline{e_k}$ on groups. Thus, the predicate and the resulting expressions are evaluated in an environment extended with the groups computed in $\mathbb{F}_3$.

Example 3.1. Consider the query `Q1` from Section 2.1 in SQL and its corresponding SQL_{Alg} expression:[4]

```
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*b2) = 2);
```

$$\gamma_{((a_1 \text{ as } a_1), a_1, \texttt{exists}(\gamma_{((a_2 \text{ as } a_2), a_2, \texttt{sum}(1+0*b_2)=2)}(t_2)))}(t_1)$$

Following the semantics of Figure 1 with $t_1 = \{\!|(a_1 : 1, b_1 : 1), (a_1 : 1, b_1 : 2), (a_1 : 2, b_1 : 3), (a_1 : 3, b_1 : 1), (a_1 : 3, b_1 : 2), (a_1 : 3, b_1 : 3)|\!\}$, the γ operator first creates the partition of t_1 according to the value of the attribute a_1:

$$\mathbb{F}_2 = [T_1, T_2, T_3] \text{ with } T_1 = \{\!|(a_1 : 1, b_1 : 1), (a_1 : 1, b_1 : 2)|\!\}, T_2 = \{\!|(a_1 : 2, b_1 : 3)|\!\} \text{ and } T_3 = \{\!|(a_1 : 3, b_1 : 1), (a_1 : 3, b_1 : 2), (a_1 : 3, b_1 : 3)|\!\}$$

The formula `exists(...)` is evaluated on each group T in $\mathbb{F}_2$ in an environment $\mathscr{E} = ([a_1, b_1], a_1, T)$. If $t_2 = \{\!|(a_2 : 7, b_2 : 7), (a_2 : 7, b_2 : 8)|\!\}$, similarly, the nested query is a γ operator that creates the partition of t_2 according to the value of the attribute a_2:

$$\mathbb{F}'_2 = [\{\!|(a_2 : 7, b_2 : 7), (a_2 : 7, b_2 : 8)|\!\}]$$

So the formula $\texttt{sum}(1 + 0 * b_2) = 2$ is evaluated in an environment $\mathscr{E}' = [([a_2, b_2], a_2, \{\!|(a_2 : 7, b_2 : 7), (a_2 : 7, b_2 : 8)|\!\}), ([a_1, b_1], a_1, T)]$ for each $T \in \mathbb{F}_2$. The evaluation of this formula will evaluate b_2 (even if it is multiplied by 0): b_2 appears twice in the first slice of the environment, so no matter T, the expression $1 + 0 * b_2$ is summed twice and $\texttt{sum}(1 + 0 * b_2) = 2$ is true for any T. This is why the entire query `Q1` returns the three grouping values for a_1: 1, 2 and 3, thus the bag $\{\!|(a_1 : 1), (a_1 : 2), (a_1 : 3)|\!\}$.

For the query `Q2`, the same reasoning holds, except that the inner formula is $\texttt{sum}(1 + 0 * b_1) = 2$. This time, the evaluation of this formula will evaluate b_1, which appears in the second slice. Thus, $1 + 0 * b_1$ is summed twice for T_1, once for T_2 and three times for T_3, which makes $\texttt{sum}(1 + 0 * b_1) = 2$ valid only for T_1. The result of the query is thus only the grouping value $a_1 = 1$, thus the bag $\{\!|(a_1 : 1)|\!\}$. ■

The semantics of formulas uses a three-valued logic where Boolean values can be *true*, *false*, or *unknown*. The value *unknown* is introduced by predicates on the `NULL` value. For example, the result

[4]For simplicity, we omit renamings that would have been added by SQL_{Coq}.

of $1 >$ `NULL` is *unknown*. The operators in this logic are noted $\wedge_3$, $\vee_3$, and $\neg_3$. They provide the maximum information, so for example *true* $\vee_3$ *unknown* = *true* and *false* $\vee_3$ *unknown* = *unknown*.

The rule for accessing an attribute a in Figure 1 looks up a value in the environment, which is traversed from the top of the stack until the attribute is found, as we have seen in the example. The same process is used for aggregates $\mathfrak{ag}$, but a group is searched for in the environment instead of a single attribute. Depending on the slice, T can be either a tuple (if it is introduced by a anything but grouping) or a list (if introduced by grouping). The well formedness of the query guaranties that the access to an attribute ($T.a$) can only occur on a tuple and not a list.

3.2 NRA^e

NRA^e [Auerbach et al. 2017a] is an extension of Nested Relational Algebra [Cluet and Moerkotte 1993], designed for optimizations. For example, there is no stratification between expressions, formulas, and queries, which allows cross level rewriting. Some optimizations strategies already exist for this language [Claußen et al. 1997, 2000; Cluet and Moerkotte 1993; Moerkotte 2020].

As suggested by the name, NRA^e supports nested data:

$$d \quad ::= \quad c \mid \{\overline{A_n : d_n}\} \mid [\overline{d_n}] \mid \texttt{left}\ d \mid \texttt{right}\ d$$

A value is either an atom (a constant), a record, a bag, or a value of type `either` (a value with a constructor `left` or `right`). A record is a mapping from a finite set of attributes to values. A large set of atoms is supported including numbers, strings, and Booleans. Values of type `either` are used to encode SQL_{Alg}'s typed null values and three-valued logic (Section 4.4).

The syntax of the language is the following:

$$\begin{array}{rcl} q & ::= & d \mid \texttt{In} \mid \boxplus q \mid q_1 \boxtimes q_2 \mid q_2 \circ q_1 \mid \chi_{\langle q_2 \rangle}(q_1) \mid \sigma_{\langle q_2 \rangle}(q_1) \mid q_1 \times q_2 \mid q_1 \texttt{ ?? } q_2 \mid q_1 | q_2 \\ & & \mid \texttt{Env} \mid q_2 \circ^e q_1 \mid \chi^e_{\langle q \rangle} \mid \texttt{group\_by}_g(\overline{a}, q) \end{array}$$

A query d returns the value d. The query `In` returns the data d it is evaluated against. The queries $\boxplus q$ and $q_1 \boxtimes q_2$ represent the application of unary operators (like negation, field access, building a singleton collection) and binary operators (like union of bags, record concatenation).

The query composition $q_2 \circ q_1$ illustrates the combinatorial nature of the semantics. It first evaluates q_1 on the input data, then uses the result of the evaluation to evaluate q_2. The query $\chi_{\langle q_2 \rangle}(q_1)$ evaluates the query q_2 on each element of the bag returned by the evaluation of q_1. The operators $\sigma_{\langle q_2 \rangle}(q_1)$ and $q_1 \times q_2$ are, respectively, selection and Cartesian product. The semantics of product use the $\oplus$ binary operator, which performs record concatenation. For example, the expression $\{a : \mathit{true}\} \oplus \{b : 3\}$ evaluates to $\{a : \mathit{true}, b : 3\}$.

The operators q_1 `??` q_2 and $q_1|q_2$ are control structures. The query q_1 `??` q_2 checks the result of running q_1 on the input data. If it is not an empty bag it returns it, otherwise it evaluates q_2 on the input data. The query $q_1|q_2$ matches the input data with `left` d and `right` d and executes either q_1 or q_2 on the data d as appropriate.

The queries `Env`, $q_2 \circ^e q_1$, and $\chi^e_{\langle q \rangle}$ manipulate the local environment ρ. `Env` returns the environment ρ, $q_2 \circ^e q_1$ updates it, and $\chi^e_{\langle q \rangle}$ iterates over it.

NRA^e also provides a $\texttt{group\_by}_g(\overline{a}, q)$ construct that evaluates q and groups the result using the values of the fields $\overline{a}$ as keys. The result is a collection of records made of the key and a field g containing the associated group. For example, $\texttt{group\_by}_g(x, d)$ where $d = [\{x : 1, y : 1\}, \{x : 1, y : 2\}, \{x : 2, y : 3\}]$ returns $[\{x : 1, g : [\{x : 1, y : 1\}, \{x : 1, y : 2\}]\}, \{x : 2, g : [\{x : 2, y : 3\}]\}]$.

$$\frac{}{\rho \vdash d_0 \ @\ d \Downarrow_a d_0}\ Constant \qquad \frac{}{\rho \vdash \mathtt{In} \ @\ d \Downarrow_a d}\ ID \qquad \frac{\rho \vdash q_1 \ @\ d_0 \Downarrow_a d_1 \quad \rho \vdash q_2 \ @\ d_1 \Downarrow_a d_2}{\rho \vdash q_2 \circ q_1 \ @\ d_0 \Downarrow_a d_2}\ Comp$$

$$\frac{}{\rho \vdash \mathtt{Env} \ @\ d \Downarrow_a \rho}\ Env \qquad \frac{\rho_1 \vdash q_1 \ @\ d_1 \Downarrow_a \rho_2 \quad \rho_2 \vdash q_2 \ @\ d_1 \Downarrow_a d_2}{\rho_1 \vdash q_2 \circ^e q_1 \ @\ d_1 \Downarrow_a d_2}\ \mathrm{Comp}^e$$

$$\frac{\rho \vdash q_1 \ @\ d \Downarrow_a d_1}{\rho \vdash q_1 | q_2 \ @\ \mathtt{left}\ d \Downarrow_a d_1}\ \mathrm{Either}_{\mathtt{left}} \qquad \frac{\rho \vdash q_2 \ @\ d \Downarrow_a d_2}{\rho \vdash q_1 | q_2 \ @\ \mathtt{right}\ d \Downarrow_a d_2}\ \mathrm{Either}_{\mathtt{right}}$$

Fig. 2. Semantics of NRA^e (excerpt).

This construct can be defined using simpler constructs of NRA^e as follows:

$$\begin{aligned}\mathtt{group\_by}_g(\overline{a}, q) \quad = \quad & \chi_{\left\langle \mathtt{In} \oplus \{g : \sigma_{\left\langle \mathtt{Env}.key = \pi[\overline{a}](\mathtt{In}) \right\rangle}(\mathtt{Env}.input) \circ^e (\{key : \mathtt{In}\} \oplus \mathtt{Env}) \right\rangle} \\ & \quad (\mathtt{distinct}(\chi_{\langle \pi[\overline{a}](\mathtt{In}) \rangle}(\mathtt{Env}.input))) \\ & \circ^e \{input : q\}\end{aligned}$$

It is easiest to understand how this definition works by proceeding backwards. The last line creates a record with a single label named *input* that contains the result of evaluating q. The $\circ^e$ expression causes this record to be used as the environment when evaluating the preceding lines.

The middle line constructs the set of distinct keys by iterating over *input*. It uses two operators: $\mathtt{distinct}(d)$ which takes a bag d and removes the duplicates, and record projection $\pi[\overline{a}](d)$ which takes a record d and returns the same record with only the specified labels $\overline{a}$. Since *input* is stored in the current environment (thanks to the third line), it can be accessed by $\mathtt{Env}.input$. Using map (χ) and project (π), we extract the keys from *input* and use `distinct` to ensure they are unique.

The last line maps over the set of distinct keys. For each one, we are building a record containing the key and a field g constructed as follow. We first extend our environment with an additional field *key* containing the current key ($\{key : \mathtt{In}\} \oplus \mathtt{Env}$). Then, we select the records in *input* matching the key ($\sigma_{\langle \mathtt{Env}.key = \pi[\overline{a}](\mathtt{In}) \rangle}(\mathtt{Env}.input)$).

The formal semantics of NRA^e is defined by a judgment $\rho \vdash q \ @\ d \Downarrow_a d'$ which means that a query q evaluated in a local environment ρ against input data d produces a value d' where the environment ρ can be any NRA^e data (*e.g.*, a record or a collection). A few rules are given in Figure 2. The complete semantics can be found in the extended version of the paper.

Compared to the original algebra, we have replaced the operator $q_1 \| q_2$ which was testing if the input was the empty collection by the operator $q_1 | q_2$ which corresponds to the pattern matching on the values `left` and `right`.

3.3 Imp

The goal of Imp, the final language we define, is to be close to the targeted runtime. It is parameterized by a data model (the constant values), the built-in operators (like addition), and the library functions (the runtime needed to execute the program). Imp can be instantiated into a subset of most imperative languages. The syntax of Imp is the following:

$$\begin{array}{lll} e & ::= & c \mid x \mid op(e) \mid f(e) \\ s & ::= & \{\ decl^* s^*\ \} \mid x := e \mid \mathtt{for}\ x\ \mathtt{in}\ e\ \mathtt{do}\ s \mid \mathtt{if}\ e\ \mathtt{then}\ s\ \mathtt{else}\ s \\ decl & ::= & \mathtt{var}\ x \mid \mathtt{var}\ x = e \\ q & ::= & \mathtt{fun}(x)\{\ s;\ \mathtt{return}\ y\} \end{array}$$

A query q is a function that takes an argument x as the input data. Its body is an imperative statement s that must define the value of the returned variable y. Statements can be assignments, loops over a collection, conditionals and blocks. A block can contain a list of variable declarations,

that can be initialized or not, followed by a sequence of statements. Finally, expressions are constants (c), variables (x), operator applications ($op(e)$), and runtime operator calls ($f(e)$).

The semantics $[\![e]\!]^{\text{imp}}(\rho)$ evaluates in an environment ρ an expression e into a value c and $[\![s]\!]^{\text{imp}}(\rho)$ evaluates statement s into a new environment ρ'. The definition is standard and given in the extended version of the paper. The only particularity is that an instantiation of Imp must provide a semantics $\{\![.]\!\}$ for the parameterized operator and library functions. The instantiation must also provide two functions $toBool(c)$ and $toList(c)$ that reify values of the language respectively into a Boolean and into a list. Assuming that all the instantiated operators are terminating, all Imp programs are terminating. For example, the semantics of the conditional ($[\![\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]^{\text{imp}}(\rho)$) has to interpret the result of the evaluation of e as a Boolean. The two expressions $op(e)$ and $f(e)$ have the same semantics. They are separate in Imp to distinguish functions that are compiled into a built-in operator in the target language, with functions that are compiled into a runtime library function. For example, the addition between two integers is compiled into a runtime library function in JavaScript, but it could be compiled into a built-in operator if we target another language that supports integers.

As our target is JavaScript code, we instantiate the Imp data model with EJson, an extended JSON with integers and functional arrays (the `null` of JavaScript does not have the same semantics of the `NULL` of SQL):

$$c \quad ::= \quad \text{string val} \mid \text{number val} \mid \text{bool val} \mid \texttt{null} \mid \{\ \overline{l_n : c_n}\ \} \mid \text{integer val} \mid [\ \overline{c_n}\ ]$$

The operators are those of the host language. For example, the operator `*` corresponds to multiplication on JavaScript numbers (IEEE754 floating point numbers). Supporting SQL requires Boolean arithmetic and string operators, as well as comparisons and access to the fields of an object.

Finally, the instantiation of Imp also comes with runtime functions that need to be implemented in JavaScript. Examples of such functions are operations on integers and functional arrays.

4 FROM SQL_{Alg} TO NRA^e

This section presents the translation between SQL_{Alg} and NRA^e. The key aspects are the following:

(1) encoding `NULL` values and three-valued logic, and the operations on them;
(2) reflecting the environments and how they are handled.

The first challenge is that NRA^e does not support three-valued logic connectives. To address this, we encode these connectives, as discussed in Section 4.2. The second challenge arises from the subtle handling of environments presented in Section 3.1. We address this by making the SQL_{Alg} environment explicit in the generated NRA^e expression (Sections 4.3 and 4.4). These challenges in the translation, presented in this section, are also reflected in the proof, as explained in Section 7.2.

SQL_{Alg} is a stratified language with multiple syntactic categories (queries, formula, and expressions), whereas NRA^e is an expression language. Similarly the SQL data are stratified but not the NRA^e ones. This complicates the translation and proofs. Notationally, we index each translation function with the syntactic category of its argument (Q for queries, f for formula, ...).

Before detailing the translation from SQL_{Alg} to NRA^e, we state the (verified) correctness theorems.

4.1 Correctness

Correctness of the translation applies only to *meaningful* SQL_{Alg} queries: queries that are well typed and well formed [Benzaken and Contejean 2019]. For simplicity, we elide this assumption in the following theorems.

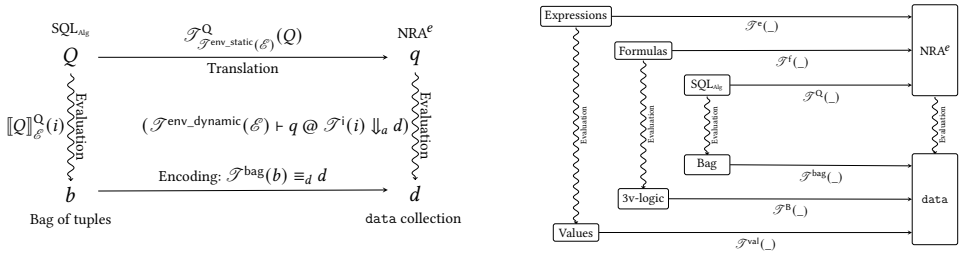


Fig. 3. SQL_{Alg} to NRA^e correctness diagram: top-level (left, Theorems 4.1 and 4.2) and internally (right, Theorems 4.2, 4.3 and 4.4).

Theorem 4.1 is the main theorem. It states the correctness of the translation from SQL_{Alg} to NRA^e: for any query Q, the evaluation of its translation to NRA^e ($\mathscr{T}^{\text{Q}}(Q)$) on the translated instance $\mathscr{T}^{\text{i}}(i)$ is equal to the translation of the result of its evaluation ($\mathscr{T}^{\text{bag}}([\![Q]\!]^{\text{Q}}(i))$).

Theorem 4.1. $\forall Q\ i, (\vdash \mathscr{T}^{\text{Q}}(Q) \ @\ \mathscr{T}^{\text{i}}(i) \Downarrow_a \mathscr{T}^{\text{bag}}([\![Q]\!]^{\text{Q}}(i)))$ □

Theorem 4.2 generalizes Theorem 4.1 to any environment, which is needed for sub-queries. As we will detail in Section 4.3, the SQL_{Alg} environment $\mathscr{E}$ can be seen has having a statically determinable part ($\mathscr{T}^{\text{env\_static}}(\mathscr{E})$, containing the groups and attribute names but not the data) that is used at compile-time, while its dynamic part actually containing the tuples of the sub-queries ($\mathscr{T}^{\text{env\_dynamic}}(\mathscr{E})$) will be available at runtime. The generalized theorem thus states that for any query Q and any environment $\mathscr{E}$, the evaluation of the translation of Q in the dynamic environment $\mathscr{T}^{\text{env\_dynamic}}(\mathscr{E})$ is equal to the encoding (using the static environment $\mathscr{T}^{\text{env\_static}}(\mathscr{E})$) of the evaluation of Q into NRA^e data model.

Theorem 4.2. $\forall Q\ \mathscr{E}\ i, (\mathscr{T}^{\text{env\_dynamic}}(\mathscr{E}) \vdash \mathscr{T}^{\text{Q}}_{\mathscr{T}^{\text{env\_static}}(\mathscr{E})}(Q) \ @\ \mathscr{T}^{\text{i}}(i) \Downarrow_a \mathscr{T}^{\text{bag}}([\![Q]\!]^{\text{Q}}_{\mathscr{E}}(i)))$ □

An alternative view to Theorem 4.2 is given in Figure 3 (left).

The translation of queries ($\mathscr{T}^{\text{Q}}$) relies on the translation of formulas ($\mathscr{T}^{f}$) and expressions ($\mathscr{T}^{e}$). Theorems 4.3 and 4.4 establish semantics preservation of these translations. To compare evaluations of SQL formulas and expressions and their corresponding NRA^e expressions, SQL Booleans and values are translated into NRA^e data ($\mathscr{T}^{\text{B}}$, and $\mathscr{T}^{\text{val}}$). Figure 3 (right) presents these theorems graphically.

Theorem 4.3. $\forall f\ \mathscr{E}\ i, (\mathscr{T}^{\text{env\_dynamic}}(\mathscr{E}) \vdash \mathscr{T}^{\text{f}}_{\mathscr{T}^{\text{env\_static}}(\mathscr{E})}(f) \ @\ \mathscr{T}^{\text{i}}(i) \Downarrow_a \mathscr{T}^{\text{B}}([\![f]\!]^{\text{b}}_{\mathscr{E}}(i)))$ □

Theorem 4.4. $\forall e\ \mathscr{E}, (\mathscr{T}^{\text{env\_dynamic}}(\mathscr{E}) \vdash \mathscr{T}^{\text{e}}_{\mathscr{T}^{\text{env\_static}}(\mathscr{E})}(e) \ @\ [] \Downarrow_a \mathscr{T}^{\text{val}}([\![e]\!]^{\text{e}}_{\mathscr{E}}))$ □

4.2 Translation of the Data Model

Database instances are the contents of the relations (tables). Since we consider only `select` queries, the database instance is constant during evaluation. For SQL_{Alg}, it is defined as a record where fields are labeled with table names and values are bags of tuples. For NRA^e, it is defined as a record where values can be any `data`. The function $\mathscr{T}^{\text{i}}$ translates database instances, mapping relation names to NRA^e's labels ($\mathscr{T}^{\text{tab}}$). Each SQL_{Alg} tuple is translated into a NRA^e record where the name of each attribute is mapped to a label ($\mathscr{T}^{\text{att}}$) and each value is translated into a NRA^e data ($\mathscr{T}^{\text{val}}$).

Value translation needs to handle `NULL`. This is done using an option type for nullable values. Following NRA^e convention, this is represented by boxing each value in a data of type `either`,

which can be `left` or `right`. A non-null value v is represented as `left` v and a null value is represented as `right ()`.

To encode the three-valued logic in NRA^e, we also use the `either` type:

$$true_3 = \texttt{left true} \qquad false_3 = \texttt{left false} \qquad unknown = \texttt{right ()}$$

We define the operators $\neg_B, \wedge_B, \vee_B$ and $\texttt{is\_true}_B$ as NRA^e expressions. For example, the $\neg_B$ operator is implemented as follows:

$$\neg_B\, q \quad = \quad (\texttt{left } (\neg\ \texttt{In}) | \texttt{right ()}) \circ q$$

This expression uses the $\circ$ operator to first evaluate q to a data d and then give d as input to $(\texttt{left } \neg\ \texttt{In} | \texttt{right ()})$. Then the matching operator $(q_1|q_2)$ returns `left` $\neg\ b$ if $d =$ `left` b otherwise it returns `right ()`. So $\neg_B\, q$ has the expected behavior of returning `unknown` if q is `unknown` and the negation of the Boolean otherwise.

Floating points and bags: an inconsistency. DBcert supports Boolean values, integers, and strings. The SQL specification also supports floating point operations. Unfortunately, however, our initial attempts at supporting them revealed that the aggregate operators `sum` and `avg` are not compatible with the set and bag semantics of SQL despite both being mainstream features of Relational Database Management Systems (RDBMSs).

Indeed, the specification requires for the aggregates `sum` and `avg` that addition is associative (and commutative), as aggregates are operating over (unordered) bags; but floating point addition is not associative. This issue is a fundamental problem with using floating point aggregate operations over unordered bags, and is a (mostly ignored) problem in real Relational Database Management Systems.

Our base compiler avoids this inconsistency by eliding support for floating point. However, to target queries based on realistic JSON databases, supporting floating point operations is important.

There are a number of possible solutions to this issue. We could acknowledge that floating point addition is indeed non-associative, and model that in the semantics. If we keep a bag semantics, this would change the semantics to be non-deterministic. If we change from using a bag semantics, we have a different infelicity to traditional semantics that would significantly inhibit query optimization.

Alternatively, we can avoid the non-associativity of floating point by using a slight-of-hand often employed in theorem proving: changing from modelling floating point numbers to modelling real numbers. This would regain associativity, but be unfaithful to our extracted implementation.

Another option is to keep modelling floating point numbers, but pretend that the `sum` and `avg` aggregate operators are associative. This keeps the model simple, but introduces (false) axioms, that need to be carefully isolated so they do not infect the rest of the verification effort.

We created a variant of our compiler that proceeds along the lines of the last option. We extended SQL_{Alg} with double precision floating point values, using Coq's native floats. We also extended the functions $\mathfrak{fn}$ with arithmetic and Boolean operations on these values, and aggregate operators $\mathfrak{ag}$ with `sum`, `max`, and `avg`. This pragmatic approach continues to model and reason about floating point numbers, while pretending that addition is associative and commutative by assuming these properties as axioms. While these axioms are technically unsound, we took great care to isolate their usage to these proofs of the floating point aggregates. Note that three other axioms about floating point numbers are also assumed, but these are all valid: associativity and commutativity of floating point maximum, and a specification for injecting positive integers into floats. These are specified as axioms since they characterize functions not implemented in Coq and only realized during extraction, however they are believed to be sound.

In addition to the care taken to isolate the use of these unsound axioms, we preserve the core version of the compiler, which does not contain these axioms (or the problematic floating point

operations), verifying the unconditional correctness of our base compiler, as described in this paper. Both versions of the compiler are provided in the artifact.

While IEEE floating point is fundamentally ill-suited for a bag semantics, we hope that future work can explore some of the other tradeoffs discussed above.

4.3 Translation of the Environment

We have seen in Section 3.1 that, during the evaluation of SQL_{Alg} nested queries, one has to know groups and data of outer queries: this is the role of the environment. This environment is implicit: it is progressively populated when traversing queries, and the correct slice in which to find the data (through attribute names) is automatically computed when needed.

NRA^e also has an ambient environment ρ. However, its manipulation is explicit: one has to store and retrieve data in it through dedicated constructs.

To faithfully capture the SQL_{Alg} semantics in NRA^e, the implicit manipulation of the SQL_{Alg} environment thus has to be made explicit at compile time. It means that the translation, in addition to reflecting the query operators (see next section), adds administrative expressions to manipulate the environment.

Runtime environment. The runtime environment to execute a NRA^e query coming from the translation of a SQL_{Alg} query mimics the SQL_{Alg} environment. It has the structure of a stack of slices, encoded as a link list in the record ρ: $\rho = \{slice : data_1, tail : \{slice : data_2, tail : ...\}\}$. The values $data_1, data_2, \ldots$ correspond to each slice computed at runtime.

NRA^e expressions for administrative steps. For this environment to be correctly handled during the evaluation of the query, the translation has to make explicit:

- how to populate ρ when traversing queries; and
- how to retrieve data in the correct slice of ρ.

For the populating part, translation will inject administrative steps to add a new slice. We have seen in Section Section 3.1 that slices can be composed of a single data or a collection. The administrative steps are respectively these two NRA^e expressions:

$$\begin{array}{lcl} \mathsf{push}_{one} & = & \{slice : [\mathtt{In}], tail : \mathtt{Env}\} \\ \mathsf{push}_{bag} & = & \{slice : \mathtt{In}, tail : \mathtt{Env}\} \end{array}$$

We remind the reader that `Env` is the expression that gives access to ρ, and `In` is the expression corresponding to the current input data. Hence both constructions add the current data on top of the current environment, with the difference that for push_{one} the current data is put in a singleton collection.

To retrieve data in the nth slice of the environment, the administrative step is simply the NRA^e expression $\mathtt{Env}.tail.\cdots.tail.slice$ where there are $n-1$ *tail* projections.

Adding administrative steps at compile time. To correctly add these steps at compile time, the translation function for queries $\mathscr{T}^{\text{Q}}_{\mathscr{A}}$ is parameterized by a *abstract translation environment* $\mathscr{A}$, which contains static information about the environment $\mathscr{E}$. Indeed, in a slice (A, G, T), A and G depend only on the structure of the query. For example, for the query $\sigma_f(\pi_{x \text{ as } a}(t))$, the top slice S of the environment in which the formula f is executed is such that $A(S) = a$ and $G(S) = []$. Therefore, we define the translation environment $\mathscr{A} = [S^a_n; ...; S^a_1]$ as a stack of static slices $S^a_i = (A_i, G_i)$.

Example 4.5. Let us give the intuition on the SQL_{Alg} queries of Example 3.1.[5]

At compile time, the translation starts with an empty translation environment.

[5]More details on this example are given at the end of the whole section.

$$
\begin{array}{lcl}
\mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(Q_1 \bowtie Q_2) & = & \mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(Q_1) \times \mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(Q_2) \\
\mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(\sigma_f(Q)) & = & \sigma_{\left\langle \mathscr{T}^{\mathrm{f}}_{(sort\ Q,\{\})::\mathscr{A}}(f) \circ^e \mathsf{push}_{one} \right\rangle}\left(\mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(Q)\right) \\
\mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(\pi_{(\overline{e_n \,\mathsf{as}\, a_n})})(Q) & = & \chi_{\left\langle \mathscr{T}^{\mathrm{Sel}}_{(sort\ Q,\{\})::\mathscr{A}}(\overline{e_n \,\mathsf{as}\, a_n}) \circ^e \mathsf{push}_{one} \right\rangle}\left(\mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(Q)\right)
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(\gamma_{(\overline{e_n \,\mathsf{as}\, a_n},\overline{b_k},f)}(Q)) & = & \text{let groups} = \chi_{\langle \mathtt{In}\cdot g\rangle}\left(\mathtt{group\_by}_g(\overline{b_k}, \mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(Q))\right) \text{ in} \\
& & \text{let filtered\_groups} = \sigma_{\left\langle \mathscr{T}^{\mathrm{f}}_{(sort\ Q,\overline{b_k})::\mathscr{A}}(f) \circ^e \mathsf{push}_{bag} \right\rangle}(\text{groups}) \text{ in} \\
& & \chi_{\left\langle \mathscr{T}^{\mathrm{Sel}}_{(sort\ Q,\overline{b_k})::\mathscr{A}}(\overline{e_n \,\mathsf{as}\, a_n}) \circ^e \mathsf{push}_{bag} \right\rangle}(\text{filtered\_groups}) \\
& & \qquad \text{where } g \text{ is a fresh label } w.r.t\ \ \overline{b_k}
\end{array}
$$

Fig. 4. Non-trivial transformations of $\mathrm{SQL}_{\mathrm{Alg}}$ queries to NRA^e.

It generates code for the outer γ by

- inserting the NRA^e code that populates the environment using the function push_{bag}; and
- calling itself recursively on the translation environment $[([a_1, b_1], a_1)]$ and the formula `exists(...)`.

In this recursive call, similarly, it generates code by using push_{bag} again and calling itself recursively on the translation environment $[([a_2, b_2], a_2); ([a_1, b_1], a_1)]$ and the formula $\mathtt{sum}(1 + 0 * b) = 2$.

Finally, the translation of b uses the translation environment to insert the correct code to retrieve data: in the case of $b = b_2$, the code is `Env.`*slice* (since b_2 is in the first slice on the translation environment); in the case of $b = b_1$, the code is `Env.`*tail.slice* (since b_1 is in the second slice). ■

Proof invariant. For the correctness statement and proof, we have to relate the $\mathrm{SQL}_{\mathrm{Alg}}$ environment with the translation and runtime environments of NRA^e. This is done through two helper functions:

- the function $\mathscr{T}^{\mathsf{env\_static}}(\bullet)$ computes the translation environment by erasing the field T from $\mathrm{SQL}_{\mathrm{Alg}}$ environment's slices:

$$
\begin{array}{rcl}
\mathscr{T}^{\mathsf{env\_static}}([]) & = & [] \\
\mathscr{T}^{\mathsf{env\_static}}((A, G, T) :: \mathscr{E}) & = & (A, G) :: \mathscr{T}^{\mathsf{env\_static}}(\mathscr{E})
\end{array}
$$

- the function $\mathscr{T}^{\mathsf{env\_dynamic}}(\bullet)$ computes the runtime environment by erasing the fields A and G, and using the *slice*/*tail* records:

$$
\begin{array}{rcl}
\mathscr{T}^{\mathsf{env\_dynamic}}([]) & = & \{\} \\
\mathscr{T}^{\mathsf{env\_dynamic}}((A, G, T) :: \mathscr{E}) & = & \{slice : \mathscr{T}^{\mathsf{bag}}(T), tail : \mathscr{T}^{\mathsf{env\_dynamic}}(\mathscr{E})\}
\end{array}
$$

These two functions are used only for specification and proofs, not during the translation.

4.4 Transformation of Queries, Formulas and Expressions

Queries. Translation of queries is denoted by $\mathscr{T}^{\mathrm{Q}}_{\mathscr{A}}(\_)$. As explained above, the translation is parameterized by a translation environment ($\mathscr{A}$). The translation of tables and set operations (`union`, `intersect` and `except`) is straightforward: they are simply translated into the same operation in NRA^e. The translation of other queries is shown in Figure 4.

The join ($\bowtie$) is translated into a Cartesian product because $\mathrm{SQL}_{\mathrm{Alg}}$ queries have by construction distinct attribute names and in this case the semantics of both operators matches.

$$
\begin{array}{lcll}
\mathscr{T}^{\mathsf{e^f}}_{\mathscr{A}}(v) & = & \mathscr{T}^{\mathsf{val}}(v) & \text{if } v \text{ is a value} \\
\mathscr{T}^{\mathsf{e^f}}_{(A,G)::\mathscr{A}}(a) & = & \texttt{first\_elt\_of}(\texttt{Env} \cdot \mathit{slice}) \cdot \mathscr{T}^{\mathsf{att}}(a) & \text{if } a \in A \\
\mathscr{T}^{\mathsf{e^f}}_{(A,G)::\mathscr{A}}(a) & = & \mathscr{T}^{\mathsf{e^f}}_{\mathscr{A}}(a) \circ^{e} (\texttt{Env} \cdot \mathit{tail}) & \text{if } a \notin A \\
\mathscr{T}^{\mathsf{e^f}}_{\mathscr{A}}(\mathfrak{fn}\ (\overline{e^f})) & = & \mathscr{T}^{\mathfrak{fn}}(\mathfrak{fn}, (\overline{\mathscr{T}^{\mathsf{e^f}}_{\mathscr{A}}(e^f)})) & \\
\mathscr{T}^{\mathsf{e^a}}_{\mathscr{A}}(\mathfrak{fn}\ (\overline{e^a})) & = & \mathscr{T}^{\mathfrak{fn}}(\mathfrak{fn}, (\overline{\mathscr{T}^{\mathsf{e^a}}_{\mathscr{A}}(e^a)})) & \\
\mathscr{T}^{\mathsf{e^a}}_{\mathscr{A}}(\mathfrak{ag}(e^f)) & = & \mathscr{T}^{\mathfrak{ag}}(\mathfrak{ag}, (\overline{\mathscr{T}^{\mathsf{f}}_{((A,G)::\mathscr{A}')}(e^f)})) \circ^{e} (\texttt{remove\_slices}\ (\mathscr{A}, e^f)) & \\
\multicolumn{4}{l}{\quad \text{where } \texttt{remove\_slices}\ (\mathscr{A}, e^f) \text{ removes the same number of slices as } \mathbb{F}_a(\mathscr{A}, e^f)}
\end{array}
$$

Fig. 5. $\mathrm{SQL}_{\mathrm{Alg}}$ Expressions translation

The translation of $\sigma_f(Q)$ needs to take into account the encoding of the $\mathrm{SQL}_{\mathrm{Alg}}$ environment in NRA^e. It is translated as a NRA^e selection over the translation of Q, $\mathscr{T}^{\mathsf{Q}}_{\mathscr{A}}(Q)$. This selection is performed over the translation of the formula f which, in order to operate over the tuples of $\mathscr{T}^{\mathsf{Q}}_{\mathscr{A}}(Q)$, is computed in a translation environment extended with the attributes introduced by Q (the *sort* of Q). An administrative step (push_{one}) ensures that the resulting NRA^e query will be evaluated in a runtime environment extended with the result of the evaluation of $\mathscr{T}^{\mathsf{Q}}_{\mathscr{A}}(Q)$.

The translation of $\pi_{(\overline{e_n\ \mathsf{as}\ a_n})}(Q)$ is similar. The difference is that it is translated into the mapping operator of NRA^e. The pairs given to this operator are computed recursively by translating each element: the function $\mathscr{T}^{\mathsf{Sel}}_{\mathscr{A}}$ translates each expression e_n and puts them in a record.

$$\mathscr{T}^{\mathsf{Sel}}_{\mathscr{A}}(\overline{e_n\ \mathsf{as}\ a_n}) = \{\overline{\mathscr{T}^{\mathsf{att}}(a_n) : \mathscr{T}^{\mathsf{e}}_{\mathscr{A}}(e_n)}\}$$

We now come to γ. We remind the reader that the query $\gamma_{(\overline{e_n\ \mathsf{as}\ a_n},\overline{b_k},f)}(Q)$ performs three successive operations on Q: it first creates groups using $\overline{b_k}$, then filters out some of these groups with respect to the formula f, finally projects over expressions $\overline{e_n}$ (giving names $\overline{a_n}$). This order is reflected in the translation. On the translation of Q, $\mathscr{T}^{\mathsf{Q}}_{\mathscr{A}}(Q)$, it first creates groups using the helper function `group_by`, adding an extra column to remember the grouping attribute thanks to the mapping $\chi_{\langle \mathtt{In}\cdot g\rangle}$. Note that the grouping expressions must be attribute names, but it does not reduce the expressiveness of $\mathrm{SQL}_{\mathrm{Alg}}$. Second, the filtering is performed, similarly as for the selection operator, except that it applies to groups, meaning that the environment has groups instead of singletons: the translation environment contains the groups $\overline{b_k}$, and the runtime environment is extended using push_{bag}. Finally, the projection is translated similarly to the projection operator, except that it operates over groups as well.

Some basic optimizations are also implemented during the translation. For example, the selection over the `true` formula (which is often introduced by the pre-processing step from SQL to $\mathrm{SQL}_{\mathrm{Coq}}$) is directly simplified:

$$
\begin{array}{lcl}
\mathscr{T}^{\mathsf{Q}}_{\mathscr{A}}(\sigma_{\mathtt{true}}(Q)) & = & \mathscr{T}^{\mathsf{Q}}_{\mathscr{A}}(Q) \\
\mathscr{T}^{\mathsf{Q}}_{\mathscr{A}}(\gamma_{(\overline{e_n\ \mathsf{as}\ a_n},\overline{b_k},\mathtt{true})}(Q)) & = & \chi_{\left\langle \mathscr{T}^{\mathsf{Sel}}_{(\mathit{sort}\ Q,\overline{b_k})::\mathscr{A}}(\overline{e_n\ \mathsf{as}\ a_n}) \circ^{e} \mathsf{push}_{bag} \right\rangle}\left(\chi_{\langle \mathtt{In}\cdot g\rangle}\left(\mathtt{group\_by}_g(\overline{b_k}, \mathscr{T}^{\mathsf{Q}}_{\mathscr{A}}(Q))\right)\right)
\end{array}
$$

Formulas. The main point in this translation is the use of three-valued logic, where Booleans are encoded with values of type `either`. Each logical connective is thus translated into a NRA^e expression implementing the connective for three-valued logic (Section 4.2).

Expressions. Figure 5 defines the translation of expressions. The translation of an attribute a reflects the use of the translation environment that we have explained: access to the correct slice in the runtime environment will be ensured by this translation. If the attribute a is in the top-slice (*i.e.*, a is in the set of labels defined in the slice), the value of the slice is extracted from the environment ($\texttt{Env} \cdot \textit{slice}$). Since elements in a slice are wrapped in a singleton collection (*c.f.*, $\texttt{push}_{one}$), the function `first_elt_of` accesses the (only) element in the slice. If the attribute a is not in the top slice, the translated expression removes the top slice ($\texttt{Env} \cdot \textit{tail}$), allowing a to be accessed from the rest of the stack; accordingly, this slice is also removed from the translation environment.

The translation of functions ($\mathfrak{fn}$) and aggregates ($\mathfrak{ag}$) requires handling `NULL` values. For some symbols, `NULL` values are absorbing elements: if any input is `NULL`, the output is also `NULL`. For example $q_1 + q_2$ is `NULL` if either q_1 or q_2 is `NULL`. Other symbols are neutral: they skip `NULL` values. For example, `sum` q will ignore any `NULL` elements in the bag returned by q. Accounting for these different behaviors correctly is not difficult, but needs to be done carefully. In addition, the translation of aggregates ($\mathfrak{ag}$) has to access the right slice in the environment. The number of slices to remove on top of the environment stack is computed using the predicate $\mathbb{F}_a(\mathscr{A}, e^f)$ which can be defined similarly to $\mathbb{F}_e(\mathscr{E}, e^f)$ (defined in Figure 1) since it does not use the component T of the slices of $\mathscr{E}$ (that is to say, $\forall \mathscr{E}, \mathbb{F}_a(\mathscr{T}^{\text{env\_static}}(\mathscr{E}), e^f) = \mathbb{F}_e(\mathscr{E}, e^f)$).

Example 4.6. Let us illustrate the translation of the outer γ in the query `Q1` (or `Q2`) from Example 3.1. This translation first produces an NRA^e expression that builds the groups according to the grouping label using `group_by` and discards the grouping keys to keep only the groups:

$$\text{groups} = \chi_{\langle \texttt{In}\cdot g\rangle}\Big(\texttt{group\_by}_g(a_1, t_1)\Big)$$

Once the groups are built, they are filtered using the formula `exists`(...). Each group is put on the environment stack using $\texttt{push}_{bag}$ for the execution of the formula. The translation of the formula is recursively done in the environment $\mathscr{A} = [([a_1, b_1], a_1)]$ reflecting the content of the stack, and `exists`(...) is translated into `count`(...) > 0:[6]

$$\text{filtered\_groups} = \sigma_{\langle (\texttt{count}(\ldots)>0)\ \circ^e\ \texttt{push}_{bag}\rangle}(\text{groups})$$

The last part of the translation is to project the parts of the groups that we are interested in. The value of groups is put on the top of the environment stack and the translation of the projection is done in the same environment $\mathscr{A} = [([a_1, b_1], a_1)]$. Since a_1 is in the top slice of $\mathscr{A}$, the access of a_1 becomes $\texttt{first\_elt\_of}(\texttt{Env} \cdot \textit{slice}) \cdot a_1$. The generated code is thus:

$$\chi_{\langle\{a_1:\texttt{first\_elt\_of}(\texttt{Env}\cdot \textit{slice})\cdot x\}\rangle}(\text{filtered\_groups})$$

■

Putting it all together yields a fully certified compiler from SQL_{Alg} to NRA^e, which handles most constructs of SQL (correlated queries, `NULL` values, most predicate, function and aggregate symbols). It enjoys two variants: one without floating point values, and one with floating point values that will be directly used in our target language, JavaScript, but under invalid assumptions reflecting its incompatibility with bag semantics.

[6]We do not detail the translation of the condition of the `exists` since it is very similar and would obscure the discourse.

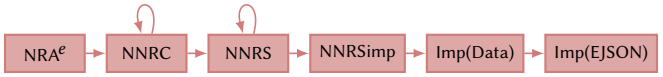


Fig. 6. Compiler Pipeline

5 FROM NRA^e TO IMP

Given the NRA^e intermediate language, we want to compile to it JavaScript. It requires (1) a paradigm switch from functional to imperative, and (2) a data representation switch from the internal data representation to JSON.

The correctness proof of this translation is the most challenging of the compilation chain. In order to handle it, we decomposed the translation into the pipeline given in Figure 6. It alternates source-to-source transformations and changes of intermediate languages where each step lowers some of the NRA^e constructs into simpler constructs that are closer to JavaScript. On the one hand, source-to-source transformations are simpler since they allow us to deal with only one semantics at a time. On the other hand, each intermediate language can enforce in its syntax and semantics some invariants which limit the scope of the proof. The alternation of these two techniques allows us to make the transformation from a language to the next one simpler.

5.1 From NRA^e to NNRC

Following the lead of Auerbach et al. [2017b], we first translate NRA^e to the named nested relation calculus (NNRC). This calculus (an extension of Van den Bussche and Vansummeren [2007]), eliminates the implicit input of NRA^e combinators, instead using explicit variables and environments. It also looks closer to a standard calculus for a functional (bag-oriented) language. As in the previous work, this translation (and the accompanying translation between the languages' associated type systems) are verified correct.

5.2 From NNRC to NNRS

NNRC, like NRA^e, is an expression oriented language: every construct returns a value. Many languages we would like to target, in contrast, are statement oriented, and evaluation proceeds via side-effects to variables. While JavaScript supports expression oriented programming, notably using first class functions, we would prefer a simpler translation that, for example, can use a `for` loop to express iterators. The next language, NNRS, is a statement oriented language: statements do not return values, but instead update the current state via (limited) side-effects. This language is inspired by the normal form in the compilation of synchrous dataflow languages that identifes functional expressions that are translated into mutable ones [Biernacki et al. 2008].

The translation from NNRC to NNRS is done in two steps. First, we define NNRC(stratified), a subset of NNRC which distinguishes between basic expressions and complex expressions, and ensures that basic expressions never have complex sub-expressions. Translation from NNRC to NNRC(stratified) hoists complex sub-expressions out of basic expressions by adding `let` construct when needed. For example, $\texttt{length}(\{(x+3)|x \in y\})$ is translated to $\texttt{let } t_1 = \{(x+3)|x \in y\} \texttt{ in length}(t_1)$. The definition of NNRC(stratified) in Coq is as a predicate on NNRC, so the translation from NNRC to NNRC(stratified) is a source-to-source transformation.

The second step of the translation is the compilation of NNRC(stratified) to NNRS where complex expressions become statements. NNRS introduces two forms of mutable variables: mutable data variables and mutable collection variables. Both of them enforce a *phase distinction*: in the first phase, the mutable variable can (only) be updated, and in the second phase, it can (only) be read. In the first phase, mutable data variables can be written (and re-written), and mutable collection variables can have elements pushed (appended). This phase distinction is enforced using a form

of `let`, called `LetMut` and `LetMutColl` respectively. They each take a variable name and two statements. They evaluate the first statement with the named variable being mutable/appendable. The value of the variable is then frozen, and can be read (but not mutated) by the second statement. Reading from a frozen data collection variable returns the accumulated bag. This phase distinction avoids *aliasing problems* by construction: once we can read a variable, we can no longer modify it.

In this language, `for` loops no longer act as implicit maps: statements do not return values. They are instead like `for` loops (over a bag) in more traditional statement oriented/imperative languages.

The translation from NNRC(stratified) to NNRS uses a form of continuation passing style, keeping track of a return continuation indicating which variable should store the return value. The `let` statements are translated into mutable `let` statements, where the final return, instead of being returned, is instead assigned to the variable. The `for` loops are translated into a definition of a mutable collection variable, with the loop nested in the first branch of the mutable collection `let` statement (before the phase barrier). The value returned by the body is pushed to the variable.

If we continue the example of the compilation of `let` $t_1 = \{(x+3)|x \in y\}$ `in length`(t_1), the corresponding NNRS code (after some simplification) is:

```
letMutColl t1 from { for (x in y) { push(t1, x + 3); } };
return (length(t1))
```

The `letMutColl t1 from { ... }; ...` constructs can update `t1` in the block following the `from` and only read its value after the `return`.

5.3 From NNRS to NNRSimp

NNRS supports a limited form of side effects. This suffices as a translation target for NNRC(stratified), but differs from target languages like JavaScript. In particular, it has three distinct namespaces, for different types of variables: mutable variables, mutable collections, and immutable variables. Also, mutable `let` statements put a variable in different environments before and after the phase barrier (moving from the mutable data or collection namespace to the "immutable" namespace).

Separating these namespaces simplifies the translation from NNRC(stratified). Notably, the different namespaces make it easy to pick fresh variables and ensure that no side effects are done on a variable after it is read. But, the benefit of the three namespaces of NNRS also comes at a cost since we want to target languages with only one namespace.

The next language in the pipeline, NNRSimp, removes the features of NNRS that were introduced only to simplify the proofs, namely the phase distinctions and the separated namespaces of NNRS. In NNRSimp, all variables are mutable (and readable). There is a single `let` construct (which introduces mutable variables), and a single assignment operator.

Similarly to the compilation from NNRC to NNRS, we first define a subset of the source language, NNRS(no-shadow), to simplify the compilation to the target language, NNRSimp. We define a predicate, named *cross-shadow-free*, that specifies what name conflicts are problematic. The intuition is that traditional shadowing is still ok, but shadowing across namespaces causes problems when they are conflated. We define a source-to-source transformation that renames variables to ensure that the result is cross-shadow-free. The translation is idempotent, and tries to rename variables minimally. Of course, it is verified to be semantics and type preserving.

Once a program is in NNRS(no-shadow) form, it is compiled to NNRSimp. The NNRS(no-shadow) language ensures that no false shadowing conflicts are introduced when the three namespaces of NNRS are collapsed into one namespace. Immutable `let` statements are re-written to be mutable `let` statements, which happen to mutate the value at most once. Mutable collection variables are encoded by initializing a mutable variable with the empty bag.

5.4 From NNRSimp to Imp

The final language in the compilation chain is Imp, which is used to handle the switch in data representation. As presented in Section 3.3, the Imp language is parameterized by its data model and the operations on it. We take advantage of that by compiling NNRSimp to Imp in two steps: first we translate to Imp(Data), which preserves the NRA^e data model (Section 3.2). We then translate Imp(Data) to Imp(EJson), which still uses Imp, but over a (slightly extended) JSON data model.

NNRSimp to Imp(Data). The main difference between NNRSimp and Imp(Data) is the lack of an Imp language construct for pattern matching on values of type `either`. This NNRSimp construct is compiled into an `if/then/else` using an Imp function `either` to test if a value is a `left` or not, and then using `getLeft` and `getRight` functions to deconstruct `either` values appropriately.

The operators of Imp(Data) are the same as the previous languages and the library functions are `either`, `getLeft`, and `getRight`. Finally, if the NRA^e `group_by` construct was preserved (and not removed as described in section 3.2), the library of Imp(Data) must provide a `group_by` function.

From Imp(Data) to Imp(EJson). Now that the query is in Imp, the last step is to switch data models, from the one of NRA^e to JSON. We use a small extension to the official JSON representation by adding a biginteger type in addition to JavaScript numbers. This is necessary to preserve the semantics for integer operations in SQL which in our formalization relies on the `Z` Coq type.

The change of data model is fundamental in that it really introduces a representation specific to the target language for the compiler (here JavaScript). In essence: collections are translated into JavaScript arrays, records are translated into JavaScript objects, and the `left` d and `right` d values of Data are encoded as JSON objects with reserved names `{ "$left": d }` and `{ "$right": d }`.

Imp(Data) functions on `left` or `right` values must be translated into equivalent Imp(EJson) functions on those objects, relying on JavaScript's ability to check if an object has a specific property.

Correctness. The shape of the correctness theorem for the translation from Imp(Data) to Imp(EJson) is worth mentioning. First it relies on a translation function from Data to EJson with good properties. Notably two Data values which translate to the same EJson have to be equal.

```
Lemma data_to_ejson_inj d1 d2: data_to_ejson d1 = data_to_ejson d2 → d1 = d2.
```

This property is fundamental to proving the main correctness theorem:

```
Lemma imp_data_function_to_imp_ejson_function_aux_correct h (d:data) (f:imp_data_function) :
  lift data_to_ejson (imp_data_function_eval h f d) =
  imp_ejson_function_eval h (imp_data_function_to_imp_ejson f) (data_to_ejson d).
```

which states that evaluating an Imp(Data) function on some data `d` and translating the result to EJson yields the same result as evaluating the corresponding Imp(EJson) function on the translation of `d` to EJson. Note that this formulation means the correctness theorem only holds for evaluating Imp(EJson) values *resulting from translating a Data value*, not for arbitrary EJson data. We believe this formulation provides the right invariant for the compiler, but this imposes that at runtime only EJson values that correspond to valid Data values are passed, a property we are careful to ensure. This additional constraint is needed (unlike our earlier translations, which do not have such a constraint), because the target data model is larger, and allows for invalid data.

6 IMPLEMENTATION

DBCert is built from a certified core in Coq with additional non-certified components in OCaml and JavaScript. We review those components here. The full development can be found in the artifact,

including some examples focused on testing the more subtle aspects of SQL's semantics [Benzaken et al. 2022a].

6.1 The Main Theorem

The certified core links the various translations between intermediate languages described in the previous sections. A theorem of semantics preservation for the full pipeline is obtained by combining individual translation proofs.

Theorem 6.1 (Semantics preservation). *Given a schema and a SQL_{Coq} query Q:*

- **if** *Q is well-formed, in the sense of Benzaken and Contejean [2019],*
- **then** *the compiler outputs an Imp query q such that, on every valid instance of the schema i, $[\![Q]\!]^{Q}(i)$ is equal to $[\![q]\!]^{imp}(\mathscr{T}^{i}(i))$ (upto bag equality).*

6.2 SQL Parser

The DBCert implementation includes a SQL parser, written in OCaml, which is used to construct an initial SQL_{Coq} abstract syntax tree (AST). The SQL grammar is written using the menhir parser generator, which can be used to generate the parser either using standard menhir or its Coq back-end [Jourdan et al. 2012] and extracting the (thus proven complete) parser.

The initial construction of the SQL_{Coq} AST performs some simple normalization of the SQL query (*e.g.* adding a `where true` if the `where` clause is missing). It ensures every intermediate expression has been named, yielding well-formed SQL_{Coq} queries. It also ensures that all attribute names are different, tagging them with the name of the relation they belong to. This step is not yet certified.

6.3 JavaScript Code Generation

From the generated Imp(EJson) code, DBCert creates a JavaScript string in two steps. First, Imp(EJson) is translated into a JavaScript AST based on the JSCert [Bodin et al. 2014] formalization of JavaScript. Then, the JSCert AST is pretty-printed as a JavaScript string.

The current DBCert produces ECMAScript 6 compliant code. It uses JavaScript blocks with `let` bindings to ensure that variable scoping in Imp blocks is being preserved in the generated code. It relies only on a small subset of ECMAScript 6 and should run in most versions of Node.js and modern browsers. The artifact has been tested with Node.js version 10.

6.4 JavaScript Runtime

Execution of SQL queries compiled to JavaScript with DBCert relies on a small run-time library written in JavaScript as well. This runtime serves two purposes: it implements runtime functions specified by the instantiation of Imp on EJson; it is used as a pre-processor for the query input in JSON, and as a post-processor for the query output.

EJson runtime. EJson supports JavaScript numbers (IEEE754 floating point numbers) and persistent (functional) arrays. The runtime provides functions to manipulate these values.

For persistent arrays, we provide two implementations. Our initial implementation was using JavaScript arrays directly, with each array operation creating a new array. But the most common operation used in the compiled code is `push` which adds one element to an array. Doing a copy of the entire array for every `push` has a strong impact on performances.

To address that issue, our current implementation uses persistent arrays where several arrays can be represented as views on the same backing data. The goal is to keep the implementation of the runtime simple and improve the performance of the `push` operation. A persistent array is simply an object with two fields: `$content`, the JavaScript array containing the data, and `$length`, an integer indicating the view of the array. With this representation, the `push` operation can be

implemented such that adding an element requires a copy of the data (`slice`) only if the size of the backing array differs from the one stored in `$length`.

Pre- and post-processors. The runtime takes care of encoding JSON values into the expected format for Imp(EJson). It includes the encoding of values which may or may not be `NULL` into the appropriate `left` and `right` representation used internally by DBCert, as described in Section 4.2 and Section 5.4. It also renames record fields to be consistent with the renaming applied when normalizing the SQL_{Coq} AST. For instance, the following JSON input:

```
{ "persons" : [ { "name" : "John" },  { "name" : null } ] }
```

is pre-processed to the following:

```
{ "persons" : array( { "persons.name" : { "left" : "John" } },
                     { "persons.name" : { "right" : null } } ) }
```

where `array` is the constructor for EJson persistent arrays.

6.5 DBCert Runner

For convenience, we provide a small Node.js script which allows one to execute queries compiled with DBCert on JSON data. This script performs the following tasks: • load a SQL query compiled to JavaScript through DBCert; • load and pre-process the database in JSON format; • execute the query; • post-process and print the query result in JSON format.

For instance, here is a (re-flowed) trace for the compilation and execution of an SQL query.

```
bash-3.2> cat tests/org2.sql
create table employees (name text, age int);
select name from employees where age > 32;
bash-3.2> ./dbcert -link tests/org2.sql
Corresponding JS query generated in: tests/org2.js
Compilation to JavaScript finished
bash-3.2> cat tests/db1.json
{ "employees": [ { "name" : "John", "age" : 34 }, { "name" : "Joan", "age" : 32 },
                 { "name" : "Jim", "age" : 33 }, { "name" : null, "age" : 35 },
                 { "name" : "Jill", "age" : null } ] }
bash-3.2> node ./dbcertRun.js tests/org2.js tests/db1.json
[{"name":"John"},{"name":"Jim"},{"name":null}]
```

7 EVALUATION AND RELATED WORK

7.1 Evaluation

We compare DBCert with AlaSQL [AlaSQL 2022], Q*cert [Auerbach et al. 2017b], and SQL.js [SQL.js 2022] which all execute SQL queries on JavaScript. AlaSQL is a popular JavaScript library with more than 13k weekly downloads on https://www.npmjs.com and 5.6k stars on GitHub. The SQL compiler from Q*cert also produces JavaScript. DBCert uses the translation from NRA^e to NNRC of this compiler. But compared to DBCert, Q*cert directly translates SQL to NRA^e and produces JavaScript code directly from NNRC. Both of these translation are not formally verified, and in particular the translation from SQL to NRA^e does not correctly handle environments. `NULL` values are not supported by this compiler. SQL.js is SQLite compiled to WebAssembly that can be then executed by the JavaScript engine. It is thus directly based on the implementation SQLite, one of the most widely deployed implementation of SQL.

We evaluate the correctness of the compiler using the queries proposed by the papers of Guagliardo and Libkin [2017] and Benzaken and Contejean [2019]. These queries have been designed to notably exercise the use of NULL and correlated queries. The difficulty with NULL is that it is generally considered as different from every values, including itself, although it is sometimes considered equal to itself. The challenge wih correlated queries is that the behavior of a subquery can depend on its evaluation context.

The benchmark contains a total fifteen queries. Four queries are covering NULL values: three proposed by Guagliardo and Libkin [2017] and one by Benzaken and Contejean [2019]. The remaining eleven queries are covering correlated queries and are proposed by Benzaken and Contejean [2019].

We take as reference the answers given by the SQL standard (when precise enough), three major RDBMSs (Oracle, PostgreSQL, SQLite), and the formal semantics of Benzaken and Contejean [2019]. On the considered queries, all of these systems agree on the expected results.

All the queries and database instances used for the evaluation are provided in the extended version of the paper and in the artifact [Benzaken et al. 2022a,b]. We refer the readers to the original papers for additional details.

The following table summarizes the results: for each compiler, we give number of valid answers per number of queries.[7]

Benchmarks	DBCert	AlaSQL	Q*cert	SQL.js
NULL	4/4	3/4	N/A	4/4
correlated queries	11/11	7/11	9/11	11/11

We note that many SQL query compilers handle these kinds of queries differently from the standard and well-established RDBMSs. These differences may lead to subtle bugs, resulting in corruption of data and processes. It is crucial to ensure the semantic correctness of compiled queries.

A preliminary performance evaluation of the generated code is presented in the extended version of the paper, but a proper evaluation is left as future work.

7.2 Challenges and Methodology

Translation from SQL_{Alg} to NRA^e. DBCert is built on top of two existing projects [Auerbach et al. 2017b; Benzaken and Contejean 2019] which made different design choices. For example, they used different techniques to implement extensible data models. In SQL_{Alg}, the formalization has two levels: a generic specification level, and a realization level that instantiates the generic components with computational definitions. In NRA^e, the data model is concrete, with extension points for external data and operators abstracted using type classes.

Both approaches have some benefits and drawbacks. To start with a project, the concrete approach of NRA^e is easier, it provides concrete objects to think, execute, and debug, whereas the abstract approach of SQL_{Alg} makes the concept more difficult to grasp and requires a realization of the data model to be able to experiment. On the other hand, the abstract approach provides a nice uniform interface to select the data model. The concrete approach necessitates splitting the code of some functions between the core of the data model and the instantiation of the extension point.

The different approaches employed created some challenges when connecting the two projects. To preserve the genericity of SQL_{Alg} with respect to the data model, the equivalence between data-models is first specified and then realized according to the concrete data model. This separation introduced by the abstract approach helped the proof development by dividing it into two phases.

Both approaches successfully enabled adding float values to the data model. The validity of the core translation needed no modification. When realizing the abstract model, the formalization

[7]The incorrect behaviors have been reported in issues 1414 and 1416 on https://github.com/agershun/alasql/.

revealed SQL's weakness when specifying the summing and averaging of float values, as discussed in Section 4.2.

Regarding proofs, the most challenging one in this part was the correctness of the translation of the environment: in SQL_{Alg}, the whole environment is present at runtime; whereas in NRA^e, the static part of the environment has been embedded in the query during the translation, and only the dynamic part is present at runtime. Relating the two in the induction was demanding. Another exigent proof was the correctness of the translation of `NULL` values and three-valued logic. We established that the chosen NRA^e operators correctly implemented the SQL_{Alg} operators. Coq was a particularly effective tool in this context, as the translation itself could be guided by the proof.

DBCert back-end. Building the DBCert back-end involved solving three major and quite different hurdles: the paradigm switch from functional to imperative languages, switching the data representation from relations to JSON, and handling variable names and scoping. We used a few specific strategies in order to deal with these difficulties while enabling proof development.

First, we used a large number of intermediate languages. This allows us to enforce some invariants in the syntax and semantics of each language, tackling each hurdle one at a time. We are satisfied by this approach, which simplifies the proofs and limits their scope. The additional translation phases do not seem to negatively affect the extracted compiler, with most of the compilation time spent on optimization. The proliferation of intermediate languages does have the disadvantage of increasing the size of the code base. But the presence of proofs simplifies maintenance since any breaking change in the code is immediately detected when compiling the corresponding proofs.

Second, we used small languages to keep them simple. This choice sometimes leads to complex encodings of some language constructs into the next one. As an example, NRA^e does not have an `if` construct. This has little impact on how we write and prove the translation: we write in Coq a function that generates a conditional using a selection (σ), prove that it behaves like a `if`, and then use it instead of a NRA^e language construct. However, this does impact the generated code, introducing redundant packing and unpacking of data in collections. These then need to be simplified through optimizations. Using small languages can thus result in a more complex compiler, despite simplifying the functions and properties of each individual language.

7.3 Related Work

The very first attempt to verify a RDBMS, using Coq, is presented in Malecha et al. [2010]. The SQL fragment considered is a reconstruction of SQL in which attributes are denoted by position. Several key SQL features, such as `group by having` clauses, quantifiers in formulas, nested, correlated queries, `NULL`'s, and aggregates, are not covered. A tool to decide SQL query equivalence was presented in Chu et al. [2017]. It relies on a K-relation [Green et al. 2007] based semantics for SQL which handles the `select from where` fragment with aggregates but does not include `having` or handle `NULL` values. Like Malecha et al. [2010], they used a reconstruction of the language, avoiding the trickier aspects of variable binding. Additionally, their semantics are not executable, making it difficult to compare it to other SQL implementations.

More closely related to our work, a translation from SQL to NRA^e was developed as part of a certified query compiler effort in Auerbach et al. [2017b]. That translation supports a realistic subset of SQL, including notably `group by having`, but it did not handle null values. It also did not include a semantics for SQL and the translation was therefore not proved correct. To the best of our knowledge the most complete formal and mechanized semantics for SQL is that developed in Benzaken and Contejean [2019], notably covering most subtleties of SQL for a practical fragment with nested correlated queries and null values. While it is executable, which means it can be

checked against actual SQL implementations, it relies on a simple interpreter with no compilation or algebraic optimization. Our work relies heavily on both projects.

Since DBCert compiles to JavaScript, it is relevant to discuss the mechanized JavaScript specification presented in Bodin et al. [2014]. While our work uses the AST provided by their work [Bodin et al. 2014] for the final code generation, attempting to prove that final part of the translation correct with respect to their semantics is left as future work.

8 CONCLUSION

We have presented a formally verified compiler from SQL to a general purpose imperative language, with a JavaScript back-end. DBCert handles a large subset of the SQL language, including nested queries and null values. Most of the compiler was proved correct using the Coq interactive theorem prover. The extracted compiler is fully functional and produces portable JavaScript code which can be executed in various environments. Importantly, one of the intermediate representations is a classic database algebra for which a large numbers of optimization techniques have been developed [Claußen et al. 1997; Cluet and Moerkotte 1993; Moerkotte 2020]. We believe this is an important step toward the development of a fully certified and practical query compiler.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

AlaSQL 2022. AlaSQL JavaScript SQL Database Library. http://alasql.org.

Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2017a. Handling Environments in a Nested Relational Algebra with Combinators and an Implementation in a Verified Query Compiler, See [Salihoglu et al. 2017], 1555–1569. https://doi.org/10.1145/3035918.3035961

Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2017b. Q*cert: A Platform for Implementing and Verifying Query Compilers, See [Salihoglu et al. 2017], 1703–1706. https://doi.org/10.1145/3035918.3056447

Véronique Benzaken and Évelyne Contejean. 2019. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019.* 249–261. https://doi.org/10.1145/3293880.3294107

Véronique Benzaken, Éveleyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2022a. *Translating Canonical SQL to Imperative Code in Coq.* https://doi.org/10.5281/zenodo.6366579

Véronique Benzaken, Évelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2022b. Translating Canonical SQL to Imperative Code in Coq – extended. *CoRR* abs/2203.08941 (2022). https://doi.org/10.48550/arXiv.2203.08941

Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES.* ACM, 121–130. https://doi.org/10.1145/1375657.1375674

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014.* 87–100. https://doi.org/10.1145/2535838.2535876

Bin Cao and Antonio Badia. 2007. SQL query optimization through nested relational algebra. *ACM Trans. Database Syst.* 32, 3 (2007), 18. https://doi.org/10.1145/1272743.1272748

Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. In *PLDI 2017* (Barcelona, Spain). ACM, New York, NY, USA, 510–524. https://doi.org/10.1145/3062341.3062348

Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. 1997. Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. In *Conference on Very Large Data Bases (VLDB).* 286–295.

Jens Claußen, Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. 2000. Optimization and Evaluation of Disjunctive Queries. *IEEE Trans. Knowl. Data Eng.* 12, 2 (2000), 238–260. https://doi.org/10.1109/69.842265

Sophie Cluet and Guido 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.

Todd J. Green, Gregory Karvounarakis, and Val 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. https://doi.org/10.1145/1265530.1265535

Akshay Grover, Jay Gholap, Vandana P. Janeja, Yelena Yesha, Raghu Chintalapati, Harsh Marwaha, and Kunal Modi. 2015. SQL-like big data environments: Case study in clinical trial analytics. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*. IEEE Computer Society, 2680–2689. https://doi.org/10.1109/BigData.2015.7364068

Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *PVLDB* 11, 1 (2017), 27–39. https://doi.org/10.14778/3151113.3151116

TU Jin-De. 2010. StreamSQL: A Query Language for Stream Data. *Computer Systems & Applications* 3 (2010), 26.

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 397–416. https://doi.org/10.1007/978-3-642-28869-2_20

Taewhi Lee, Moonyoung Chung, Sung-Soo Kim, Hyewon Song, and Jongho Won. 2016. Partial Materialization for Data Integration in SQL-on-Hadoop Engines. In *6th International Conference on IT Convergence and Security, ICITCS 2016, Prague, Czech Republic, September 26, 2016*. IEEE Computer Society, 1–2. https://doi.org/10.1109/ICITCS.2016.7740361

J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a Verified Relational Database Management System. In *ACM Int. Conf. POPL*. https://doi.org/10.1145/1706299.1706329

Guido Moerkotte. 2020. *Building Query Compilers*. Univ. Mannheim. https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). 2017. *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM. http://dl.acm.org/citation.cfm?id=3035918

Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD Conference*. ACM, 1907–1922. https://doi.org/10.1145/2882903.2915244

Avraham Shinnar, Jérôme Siméon, and Martin Hirzel. 2015. A Pattern Calculus for Rule Languages: Expressiveness, Compilation, and Mechanization. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 542–567. https://doi.org/10.4230/LIPIcs.ECOOP.2015.542

SQLAlchemy 2021. SQL Alchemy: The Python SQL Toolkit and Object Relational Mapper. https://www.sqlalchemy.org.

SQL.js 2022. SQLite compiled to JavaScript. https://sql.js.org/.

Jeffrey D. Ullman. 1982. *Principles of Database Systems, 2nd Edition*. Computer Science Press.

Jan Van den Bussche and Stijn Vansummeren. 2007. Polymorphic type inference for the named nested relational calculus. *Transactions on Computational Logic (TOCL)* 9, 1 (2007). https://doi.org/10.1145/1297658.1297661