

# Q\*cert: A Platform for Implementing and Verifying Query Compilers

Joshua S. Auerbach, Martin Hirzel, Louis Mandel,  
Avraham Shinnar & Jérôme Siméon  
IBM Research  
1101 Kitchawan Rd  
Yorktown Heights, NY 10598

## ABSTRACT

We present Q\*cert, a platform for the specification, verification, and implementation of query compilers written using the Coq proof assistant. The Q\*cert platform is open source and includes some support for SQL and OQL, and for code generation to Spark and Cloudant. It internally relies on familiar database intermediate representations, notably the nested relational algebra and calculus and a novel extension of the nested relational algebra that eases the handling of environments. The platform also comes with simple but functional and extensible query optimizers.

We demonstrate how the platform can be used to implement a compiler for a new input language or develop new optimizations that can be formally verified. We also demonstrate a web-based interface that allows the developer to explore various compilation and optimization strategies.

## 1. INTRODUCTION

In application domains where safety, security or privacy are essential, formal methods are an important part of the development process. While some effort has been made in formally proving correct traditional compilers [10], this question has received less attention in database systems in general and for query compilers in particular.

We demonstrate Q\*cert, a platform for the development and the formal verification of query compilers which is built using the Coq proof assistant [8]. We show how modern theorem proving technology can be used to specify and mechanically check for correctness a large part of the query compilation pipeline, resulting in strong correctness guarantees for the final compiler.

While the platform is targeting primarily query compilers built outside of traditional database management systems (e.g., for DSLs or language-integrated queries [12]), we believe the experience can be of interest to both database theorists and practitioners.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA

© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...\$15.00

DOI: <http://dx.doi.org/10.1145/3035918.3056447>

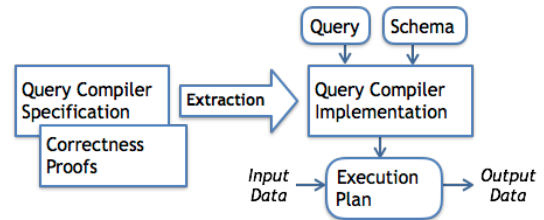


Figure 1: Query Compiler Generation

The source code for the compiler framework<sup>1</sup> and an on-line version of the demo<sup>2</sup> are publicly available on github.

## 2. THE Q\*CERT PLATFORM

Figure 1 shows the general idea behind the approach. The query compiler is first described in a high-level specification language that includes the ability to state and mechanically check proofs of correctness, which in our case is the Coq proof assistant. The compiler itself is then automatically generated from that specification using Coq’s *extraction* mechanism [11]. It ensures that it precisely matches the specification. Once extracted, that compiler behaves as any other query compiler, taking a query and a database schema and producing code for execution.

Q\*cert is built around a rich data model and type system, which include support for atomic types, records, collections, and objects (with a type hierarchy), making it suitable for a variety of query languages. At its core, it uses NRA<sup>e</sup> [1], a conservative extension of the nested relational algebra (NRA) [7]. This algebra is designed so that known results and optimizations on NRA can be applied while also facilitating compilation of languages with complex manipulating environments.

The Q\*cert platform comes with built-in support for several input query languages, including subsets of SQL and OQL, as well as JRules [3], a domain-specific rules language. The compiler can emit local JavaScript or Java code, and distributed code for Spark [15] and the Cloudant NoSQL database [6]. The complete compilation pipeline is given in Figure 2. Source languages are listed on the left of the figure, and target languages are listed on the right. Each box in the figure corresponds to an intermediate language. Arrows between two boxes are existing translations and loops

<sup>1</sup><https://github.com/querycert>

<sup>2</sup><https://querycert.github.io/>

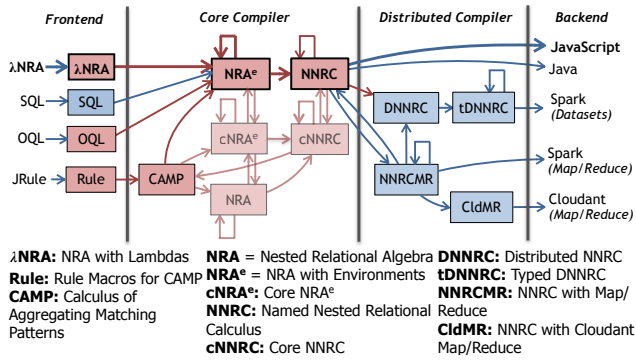


Figure 2: Q\*cert Compilation Pipeline

over a single box are existing optimizers. The part of that pipeline in red in Figure 2 comes with proofs of correctness.

While verification techniques have greatly matured,<sup>3</sup> most applications to databases have focused on query languages formalization [2, 14]. One notable exception is the Coko-Kola project [5] which used the Larch theorem prover in the context of a rule-based compiler<sup>4</sup>. By leveraging recent theorem proving advances, we were able to formally verify a much larger part of the query compilation pipeline.


### 3. AUDIENCE EXPERIENCE

We use  $NRA^\lambda$ , a simple surface query language, to demonstrate the extensibility of the platform and illustrate its internals. We show how  $NRA^\lambda$  can be integrated into the compiler and how to add (and prove) new optimizations. Finally, we demonstrate a web-based interface designed to test and debug the resulting compiler.

#### 3.1 Compiling $NRA^\lambda$

$NRA^\lambda$  [1] is a simple language of relational operators written as functions with closures, in a style similar to that of C# Linq [12] or Spark's RDD [15] operations. The following example returns the persons whose age is less than 30 from the union of two collections:

```
Persons1.union(Persons2).filter{p => p.age < 30}
```

Figure 3 sketches the different code components required to integrate  $NRA^\lambda$  into the compiler. We are going to explain them in turn. A full version of the code in HTML form can be found by clicking on the flower symbol .

In order to add a new language in Q\*cert, the first step is to define its Abstract Syntax Tree (AST). In Coq, an AST can be defined with an inductive type. The AST for  $NRA^\lambda$  is given in Figure 3 (a) as the type `lnra`. The AST nodes are prefixed with `LNRA`, and include access to input tables (`LNRAConst`), constants (`LNRAConst`), predefined unary (`LNRAUnop`) or binary (`LNRABinop`) operators which notably include comparisons, and relational operators: `map` (`LNRAMap`), `filter` (`LNRAFilter`) and `product` (`LNRAProduct`). Finally, a separate type called `lnra_lambda` with a single AST node (`LNRALambda`) is used to represent lambda terms.

<sup>3</sup>A review of the state of the domain can be found in [9].

<sup>4</sup><http://www.cs.brandeis.edu/~cokokola/>

For the frontend part of the compiler, the AST simply reflects the structure of the original query. In our example, the AST is as follows:

```
Definition demo_example :=
  LNRAFilter (LNRALambda "p"
    (LNRABinop ALt
      (LNRAUnop (ADot "age") (LNRAVar "p"))
      (LNRAConst (dnat 30))))
  (LNRABinop AUnion (LNRAConst "Persons1")
    (LNRAConst "Persons2")).
```

The second step to add a new language is to define its semantics using an evaluation function. For  $NRA^\lambda$ , the signature of that eval function is given in Figure 3 (c). It takes a global environment (`global_env`) for the input tables, a local environment (`env`) for the variables that are in scope (i.e., bound in lambda terms), and an AST node (`op`). The function returns an optional value (of type `option data`) as output, accounting for the fact that some queries may not be well-typed and fail. The evaluation function simply matches the operator against each possible AST node, evaluates its operands as needed and applies the semantics for that operator. In the case of lambda terms, the evaluation function (`lnra_lambda_eval`) takes an additional input (`d` of type `data`) corresponding to the value to be bound to the declared variable in the lambda term.

Once the language has been defined, it can be connected to the rest of the compiler by translating it to one of the existing intermediate languages. The initial target mostly commonly the nested relational algebra, which also serves as the main representation for optimization, but it can be any other available intermediate language. We use  $NRA^e$ , a variant of the nested relational algebra from [7] with a combinators semantics which is described in more details in [1]. For  $NRA^\lambda$ , the translation function's signature is given in Figure 3 (b), taking an operator (`op`) of type `lnra` as input and returning an equivalent AST of type `nraenv` for  $NRA^e$ . As an example, filter nodes in  $NRA^\lambda$  are naturally translated to selection operators in  $NRA^e$ .

So far this code could have been written in any programming language, but using Coq also allows the developer to write the correctness statement for the translation from  $NRA^\lambda$  to  $NRA^e$  as show in Figure 3 (d). The theorem states that in the context of a global environment: for every local environment `env`, every lambda term `lop`, and every input data `d`, evaluating the lambda term in that environment over value `d` gives the same result as evaluating the translation of that lambda term to  $NRA^e$  (`nraenv_eval` is the evaluation function for  $NRA^e$ ). The theorem is accompanied by a proof by induction (elided from the text for space reasons), which is mechanically verified by the Coq proof assistant.

Put together, the code fragments from Figure 3 represent only about 500 lines of code and are sufficient to implement *and verify correct* the  $NRA^\lambda$  box on the upper left of Figure 2 along with the translation from it to the  $NRA^e$  box.

For the interested audience, the demonstration will also include an on-the-fly extension to the AST for  $NRA^\lambda$  (e.g., with a flat-map operator), translation to  $NRA^e$ , and a short interactive session with the Coq proof assistant to verify that extension correct.

```

Inductive lnra : Set :=
| LNRAVar : string -> lnra
| LNRATable : string -> lnra
| LNRAConst : data -> lnra
| LNRABinop : binOp -> lnra -> lnra -> lnra
| LNRAUnop : unaryOp -> lnra -> lnra
| LNRAMap : lnra_lambda -> lnra -> lnra
| LNRAMapConcat : lnra_lambda -> lnra -> lnra
| LNRAProduct : lnra -> lnra -> lnra
| LNRAFilter : lnra_lambda -> lnra -> lnra
with lnra_lambda : Set :=
| LNRALambda : string -> lnra -> lnra_lambda
.

```

(a) Abstract Syntax Tree ❄

```

Context (global_env:list (string*data)).
Fixpoint lnra_eval (env: bindings) (op:lnra)
: option data :=
  match op with
  | LNRAVar x => edot env x
  | LNRATable t => edot global_env t
  ...
with lnra_lambda_eval (env:bindings)
                    (lop:lnra_lambda)
                    (d:data)
: option data :=
...

```

(c) Semantics with eval ❄

```

Fixpoint nraenv_of_lnra (op:lnra) : nraenv :=
  match op with
  | LNRAVar x => NRAEnvUnop (ADot x) NRAEnvEnv
  | LNRATable x => NRAEnvGetConstant x
  ...
  | LNRAFilter lop1 op2 =>
    NRAEnvSelect (nraenv_of_lnra_lambda lop1)
                  (nraenv_of_lnra op2)
  end
with nraenv_of_lnra_lambda (lop:lnra_lambda) : nraenv :=
...

```

(b) Translation to Algebra ❄

```

Context (global_env:list (string*data)).
Theorem nraenv_of_lnra_lambda_correct :
  forall env:bindings, forall lop:lnra_lambda, forall d:data,
    lnra_lambda_eval h global_env env lop d =
      nraenv_eval h global_env
                  (nraenv_of_lnra_lambda lop) (drec env) d.
Proof.
  destruct lop.
  revert env s.
  lnra_cases (induction 1) Case;
  ...
Qed.

```

(d) Translation Correctness ❄

Figure 3: Compiling NRA<sup>λ</sup>

## 3.2 Adding New Optimizations

Once a surface language has been added to the compiler, a natural next step is to allow new rewrites on a subsequent intermediate language representation for optimization purposes. Both the NRA<sup>e</sup> and NNRC intermediate languages come with a built-in optimizer that has been designed to facilitate extensibility.

A natural optimization to consider for our previous example is simply to push the filter through the union. Similarly to a rule-based approach [4, 13, 5], a new optimization can be expressed in Q\*cert as a rewrite over the algebraic plan. As an example, this is the implementation of the distributivity law for selection over union:

```

Definition select_union_distr_fun q :=
  match q with
  | NRAEnvSelect q0 (NRAEnvBinop AUnion q1 q2) =>
    NRAEnvBinop AUnion
      (NRAEnvSelect q0 q1) (NRAEnvSelect q0 q2)
  | _ => q
  end.

```

The optimizer itself is written in a functional style and applies actual pattern-matching (using `match q with...`) to check that the query plan has the shape required for the rewrite (here a selection over a union). If the query plan matches, it is rewritten accordingly. The correctness of this function can also be expressed and proved in Coq:

```

Lemma select_union_distr_fun_correctness q:
  select_union_distr_fun q ≡ q.
Proof.
  Hint Rewrite select_union_distr : envmap_eqs.
  prove_correctness q.
Qed.

```

This lemma states that for all query plans `q`, applying the

function `select_union_distr_fun` returns an equivalent query. The proposition is followed by a proof script that is mechanically checked. The proof relies on an automated proof tactic `prove_correctness` using the lemma `select_union_distr` as a hint, which stands for the distributivity law of selection over union and can be stated as follows ❄:

```

Lemma select_union_distr q0 q1 q2 :
  σ⟨ q0 ⟩(q1 ∪ q2) ≡ σ⟨ q0 ⟩(q1) ∪ σ⟨ q0 ⟩(q2).
Proof. ... Qed.

```

Once created, the optimization and corresponding proof needs simply be registered with the overall optimizer whose proof of correctness is generic (i.e., the global proof for the optimizer only relies on the proof of local correctness for new rewrites). The optimization is then available and as we will see can be turned on and off by the developer once the compiler is extracted.

## 3.3 Web Interface

Extracting the compiler itself from its Coq specification results in an executable engine that can be used from the command-line or within a Web browser. For ease of demonstration, we will mostly illustrate the extracted compiler through that Web interface. Two important features of the interface for compiler development are:

- Since the Q\*cert compilation pipeline is extensive (see Figure 2), the interface allows developer to select input and target languages and choose the specific compilation paths. This is done through a simple graphical representation<sup>5</sup> illustrated in Figure 4. The query builder shows all existing intermediate languages at

<sup>5</sup>Snapshots for the Web user interface as based on an early, but functionally complete, prototype.

the bottom, and the current compilation path at the top: in our example the developer selected  $NRA^\lambda$  as source and JavaScript as target and the rest of the compilation path is filled in automatically.

- The ability to examine and select existing optimizations as illustrated in Figure 5. The interface provides links to the rewrites specification in Coq and optimizations can be selected or removed. The figure highlights in blue the optimization for distributing filter over union that was discussed earlier. As the compiler runs, the optimizer produces traces to help identify which specific optimizations are being used and in which order.

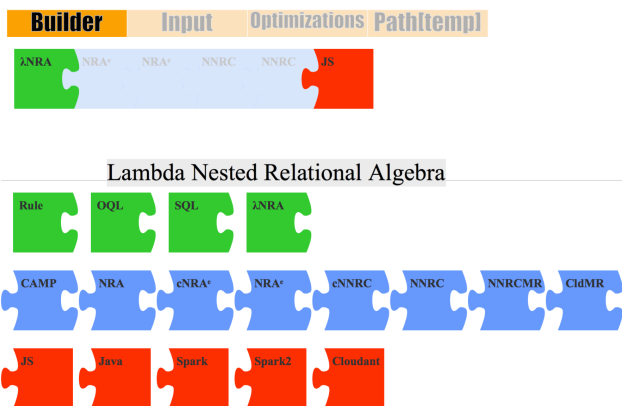


Figure 4: Compilation Path Builder

Builder	Optimizations	Input	Compile
nra	nraenv core	nraenv	nnrc core
(nraenv) head	(nraenv) tail		nnrc

Currently selected optimizations (10)	Available optimizations (92)
These optimizations will be batched in 5 iterations	
app/app ✖ ⚙	and commute ⚙
app-env/app-env ✖ ⚙	select/and swap ⚙
app-env/app body ✖ ⚙	select/select fusion ⚙
app/app-env body ✖ ⚙	select/union distr ⚙
map/id ✖ ⚙	dot/concat/rec rig <small>Pushes selection through union</small> ⚙
product/singleton singleton ✖ ⚙	dot/concat/rec left dup ⚙
product singleton right ✖ ⚙	flatten/coll ⚙
product singleton left ✖ ⚙	concat/nil right ⚙
map/coll ✖ ⚙	concat/nil left ⚙
map/map ✖ ⚙	dot/concat/rec right ⚙
	dot/concat/rec left ⚙
	merge-concat/nil right ⚙
	merge-concat/nil left ⚙
	map/id ⚙

Figure 5: Optimizer Tuning

Once the compilation path is defined and the set of optimizations selected, the user can write a query in the expected source language and compile it. Additional functionalities for the interface include:

1. Inspection of the intermediate queries for each language used during compilation from source to target.
2. Ability to specify the schema in either JSON form or SQL form.

3. Ability to load data in either JSON form or tabular form and call the evaluation function for the various intermediate languages (e.g., the one used for  $NRA^\lambda$ ) in order to check the result of a query.
4. Ability to execute code emitted for the JavaScript target. Additional query runners for the other target languages are available from the command line.

Examples to illustrate the different input query languages (SQL, OQL,  $NRA^\lambda$ , and JRule) will be part of the demo, including TPC-H queries in the SQL case.

Participants at the demonstration will be encouraged to challenge the Q\*cert team with their favorite query optimizations or rewrites!

## 4. REFERENCES

- [1] J. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In *ACM SIGMOD*, 2017.
- [2] V. Benzaken, E. Contejean, and S. Dumbrava. A Coq formalization of the relational data model. In *European Symposium on Programming (ESOP)*, 2014.
- [3] J. Boyer and H. Mili. *Agile Business Rule Development*. Springer, 2011.
- [4] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita, and M. Muralikrishna. The architecture of the EXODUS extensible DBMS. In *Workshop on Object-Oriented Database Systems*. Springer Berlin Heidelberg, 1991.
- [5] M. Cherniack and S. B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *ACM SIGMOD*, 1996.
- [6] Anatomy of the Cloudant DBaaS, 2015. <https://cloudant.com/CloudantTechnicalOverview.pdf>.
- [7] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Workshop on Database Programming Languages (DBPL)*, pages 226–242, 1993.
- [8] The Coq proof assistant reference manual, v. 8.4pl41.
- [9] X. Leroy. Desperately seeking software perfection. [www.lip6.fr/colloquium/Leroy-2015-10-20.pdf](http://www.lip6.fr/colloquium/Leroy-2015-10-20.pdf).
- [10] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [11] P. Letouzey. Extraction in Coq: An overview. In *Proceedings of the 4th Conference on Computability in Europe: Logic and Theory of Algorithms*, 2008.
- [12] E. Meijer. The world according to LINQ. *Communications of the ACM*, 54(10):45–51, 2011.
- [13] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *ACM SIGMOD*, pages 39–48, 1992.
- [14] A. Shinnar, J. Siméon, and M. Hirzel. A pattern calculus for rule languages: Expressiveness, compilation, and mechanization. In *European Conference for Object-Oriented Programming*, 2015.
- [15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.