

# Experience Report: Prototyping a Query Compiler Using Coq (Artifact)

JOSHUA S. AUERBACH, IBM Research  
MARTIN HIRZEL, IBM Research  
LOUIS MANDEL, IBM Research  
AVRAHAM SHINNAR, IBM Research  
JÉRÔME SIMÉON, IBM Research

---

The artifact accompanying the ICFP 2017 paper entitled *Experience Report: Prototyping a Query Compiler Using Coq* is available at <http://dx.doi.org/10.1145/3110253>. It is distributed as a virtual appliance using the Open Virtualization Format. It is a Linux image containing:

- the paper: `~/icfp17.pdf`
- the source code of the Q\*cert query compiler in Coq: `~/qcert`
- some examples: `~/samples`
- the executable binary of the compiler: `qcert`
- the dependencies needed to rebuild the compiler (Coq, OCaml, ...)

In the following, we explain how to launch the virtual machine and execute the query compiler on the example used in the paper. Then, we delineate which parts of the paper are covered by the artifact. Finally, we detail how to reconstruct the artifact.

General instructions on how to use the Q\*cert compiler are available in `~/qcert/README.md`.

## 1 GETTING STARTED

Launch the virtual machine `icfp17.voa` using a virtualization player like VirtualBox (<https://www.virtualbox.org>). The virtual machine was tested with VirtualBox version 5.1 ([https://www.virtualbox.org/wiki/Download\\_Old\\_Builds\\_5\\_1](https://www.virtualbox.org/wiki/Download_Old_Builds_5_1)).

Log in to the machine:

- username: `qcert`
- password: `quercert`

Remark: The virtual machine is running a ssh server. So you should be able to connect to it from the host machine (use `ip addr` in the virtual machine to get the address of the machine).

Once logged in to the machine, we can check that the Q\*cert compiler is installed with the following command:

```
qcert -help
```

This should display the command line options for the compiler.

## 2 COMPILE AND EXECUTE A QUERY STEP BY STEP

We detail step by step how to compile the query `~/samples/camp_rule/icfp17.camp_rule` presented in the paper. Then, we show how to use the Makefile to simplify the compilation.

In the following, we will suppose that the current working directory is `~/samples/camp_rule/`:

```
cd ~/samples/camp_rule/
```

## 2.1 Compile a query

To compile the file `icfp17.camp_rule`, we can simply execute:

```
qcert icfp17.camp_rule
```

It displays the compilation path used:

Compiling from `camp_rule` to `js`:

```
camp_rule -> camp -> nraenv -> nraenv -> nnrc -> nnrc -> js
```

and produces the file `icfp17.js`. The transitions `nraenv -> nraenv` and `nnrc -> nnrc` correspond to optimization phases.

In the paper, we are targeting Cloudant. So, to specify the compilation chain presented in the paper (see Figure 2), we can use the options `-exact-path`, `-source`, `-path` and `-target`:

```
qcert -exact-path \  
-source camp_rule \  
-path camp -path nraenv -path nraenv -path nnrc -path nnrc \  
-path nnrcmr -path nnrcmr -path clmgr \  
-target cloudant \  
icfp17.camp_rule
```

It produces the Cloudant query in the file: `icfp17_cloudant_design.json`.

With this combination of options, all the compilation paths presented in Figure 3 can be reproduced. The supported languages are: `camp_rule`, `camp`, `tech_rule`, `designer_rule`, `oql`, `sql`, `lambda_nra`, `nra`, `nraenv`, `nraenv_core`, `nnrc`, `nnrc_core`, `nnrcmr`, `clmgr`, `dnnrc`, `dnnrc_typed`, `js`, `java`, `spark_rdd`, `spark_dataset` and `cloudant`.

As explained in Section 5.3 of the paper, instead of explicitly giving the compilation path, the compiler can compute it automatically. Therefore, the previous command can be simplified into:

```
qcert -source camp_rule -target cloudant icfp17.camp_rule
```

By using the option `-emit-all`, it is possible to observe the query at each intermediate step of the compilation chain. The option `-emit-sexp-all` produces the same intermediate queries but in S-expression form (see Section 4).

```
qcert -source camp_rule -target cloudant icfp17.camp_rule \  
-emit-all -emit-sexp-all
```

The produced files contain the compilation step in their name.

All these intermediate files can be removed with the following command:

```
rm icfp17_camp* icfp17_rule*
```

## 2.2 Execute a query

Now that we have seen how to compile a query, we are going to see how to execute the generated code.

In order to perform the evaluation, we will need some additional information that is going to be provided in the JSON file `~/samples/io/icfp.io`.

This file contains a schema which is a set of information related to typing information. It contains in particular the subtyping relation that can be needed at runtime to compute the value of `instanceof`. (Some backends also need this typing information at compile time.)

The file contains also input which is the input data on which the query is going to be executed. The field `partitionedInput` is an alternative format for the input data that can be partitioned according to the objects type.

Finally, the file contains output the expected result by the execution of the query by be able to check the result of the execution.

**2.2.1 Evaluation using the reference semantics.** As explain in the paper, the reference semantic of each intermediate language is given by an evaluation function. We can then evaluate the queries using our reference semantics and check that at each step we obtain the same result.

In order to evaluate the query with Q\*cert, we need to provide the `.io` file to qcert and use the option `-eval-all` to ask for the evaluation. Therefore the command to evaluate the query `icfp17.camp_rule` is:

```
qcert -source camp_rule -target cloudant icfp17.camp_rule \  
-io ../io/icfp17.io -eval-all
```

It produces a JSON file `icfp17_rule_camp*.json` for each intermediate step containing the result of the evaluation. Here, with the input data given in the `io` file, the expected result is a collection containing `[22.5]` (that is `[ [22.5] ]`).

**2.2.2 Execute in JavaScript.** We are now going to see how to evaluate the generated JavaScript code produced by the compiler. But first we have to be sure that you have the generated JavaScript code:

```
qcert -source camp_rule -target js icfp17.camp_rule
```

In order to evaluate the JavaScript file `icfp17.js` produced, we use a runner provided in `~/qcert/samples`. It is a Java program relying on Nashorn, the JavaScript engine of Java.

The command to launch the runner is:

```
java \  
-cp $HOME/qcert/samples/bin:$HOME/qcert/lib/gson-2.7.jar \  
testing.runners.RunJavascript \  
-input ../io/icfp17.io -runtime ../harness.js icfp17.js
```

This program executes the JavaScript code on the input data and compare it with the expected result.

You can notice in the command line that the runner is parameterized by `icfp17.js` the generated query, `../io/icfp17.io` containing the subtyping relation and the input data. It is also parameterized by `../harness.js` a runtime library containing the code of the operators like `distinct` or `flatten`.

**2.2.3 Execute in Cloudant.** Cloudant (<https://cloudant.com>) is a Cloud database. We then first need to create a Cloudant instance in the Cloud:

1. Create a Bluemix account: <https://console.ng.bluemix.net> and login.
2. On the top right of the Bluemix dashbord click on Catalog.
3. Search for Cloudant and click on Cloudant NoSQL DB.
4. On the Cloudant NoSQL DB page click on Create at the bottom right.
5. On the left of the page click Service credentials.
6. Click View credentials on the right.
7. Copy the credentials in the file `~/cred.json` in the virtual machine.

At this point, we have a new instance of Cloudant in the Cloud and its credentials. To access the graphical interface of Cloudant, we can go to [https://\\$USERNAME.cloudant.com/dashboard.html](https://$USERNAME.cloudant.com/dashboard.html) where `$USERNAME` should be replace by the Cloudant username given in the credentials.

Let us regenerate the Cloudant code. As for the JavaScript backend, the generated Cloudant code needs a runtime library. This library has to be included directly in the generated Cloudant code. We therefore have to regenerate the Cloudant code specifying the runtime library to included with the option `-js-runtime` and to option `-io` for the subtyping relation:

```
qcert -source camp_rule -target cloudant icfp17.camp_rule \  
-js-runtime ../harness.js -io ../io/icfp17.io
```

It produces a file `icfp17_cloudant_design.json` with the runtime library included.

In order to deploy and test the generated code, we are using the Cloudant runner also provided in `~/qcert/samples`. It can be launch with to following command where `$USERNAME` and `$PASSWORD` should be replaced by Cloudant username and password provided in the credentials.<sup>1</sup>

```
java \
-cp $HOME/qcert/samples/bin:$HOME/qcert/lib/commons-codec-1.6.jar:
    $HOME/qcert/lib/gson-2.7.jar:$HOME/qcert/lib/commons-logging.jar:
    $HOME/qcert/lib/httpcore-4.3.2.jar:$HOME/qcert/lib/commons-collections-3.2.2.jar:
    $HOME/qcert/lib/cloudant-client-1.2.3.jar:$HOME/qcert/lib/httpclient-4.3.4.jar \
-Dcloudant_user=$USERNAME \
-Dcloudant_password=$PASSWORD \
testing.runners.RunCloudant \
-input ../io/icfp17.io icfp17_cloudant_design.json
```

By default, this runner delete the databases after the execution. To keep the databases in Cloudant to be able to observe them after the execution of the test, we can add the option `'-keep-db'`:

```
java \
-cp $HOME/qcert/samples/bin:$HOME/qcert/lib/commons-codec-1.6.jar:
    $HOME/qcert/lib/gson-2.7.jar:$HOME/qcert/lib/commons-logging.jar:
    $HOME/qcert/lib/httpcore-4.3.2.jar:$HOME/qcert/lib/commons-collections-3.2.2.jar:
    $HOME/qcert/lib/cloudant-client-1.2.3.jar:$HOME/qcert/lib/httpclient-4.3.4.jar \
-Dcloudant_user=$USERNAME \
-Dcloudant_password=$PASSWORD \
testing.runners.RunCloudant \
-input ../io/icfp17.io -keep-db icfp17_cloudant_design.json
```

### 2.3 More example

Additional examples are given in `~/samples/camp_rule`. For each file, the corresponding source code and SemRule code are available respectively in the directories `~/samples/def1` and `~/samples/tech_rule`.

A Makefile is provided in the directory `~/samples`:

```
cd ~/samples
```

The target compile of the Makefile compiles all the examples in JavaScript and Cloudant.

```
make compile
```

The generated JavaScript files are produced in the directory `~/samples/js` and the Cloudant ones are in `~/samples/cloudant`.

The `.io` files allowing to test the examples are available in `~/samples/io`. The target `js_run` executes all the examples using the JavaScript code:

```
make js_run
```

In order to execute the examples with the Cloudant backend, the Makefile must be modified to include the Cloudant username and password. Then the target `cloudant_run` executes all the examples using the Cloudant code:

```
make cloudant_run
```

---

<sup>1</sup>The classpath should be on one line.

## 2.4 Alternative input language

To write examples, it is possible to use other input languages than Rule. The example of the paper in OQL is provided in `~/samples/oql/icfp17.oql`.

The compilation of OQL files is similar to the compilation of Rule files. The only difference is that the argument of the `-source` option is `oql`. So, in the directory `~/samples/oql` the file `icfp17.oql` can be compiled as follows:

```
qcert -source oql -target cloudant icfp17.oql
```

The query can also be evaluated with the reference semantics as follows:

```
qcert -source oql -target cloudant icfp17.oql -io ../io/icfp17oql.io -eval-all
```

Finally, to execute the query in JavaScript and on Cloudant, we can use the Makefile:

```
make run_js_icfp17oql run_cloudant_icfp17oql
```

## 3 RELATIONSHIP WITH THE PAPER

The paper *Experience Report: Prototyping a Query Compiler Using Coq* presents how a research team at IBM used the Coq proof assistant to prototype a query compiler for the IBM product *Operational Decision Manager: Decision Server Insights* (ODM Insights).

Parts of the code belongs to a commercial product and cannot be made available. We are going through the different sections of the paper to present what is part of the artifact.

Section 2 of the paper presents the general architecture of the compiler. In particular, Figure 2 shows that JRules programs are first parsed by ODM Insights into an internal format called SemRule and then translated into CAMPRule.

SemRule being part of ODM Insights, this first translation from JRules to CAMPRule is not provided in the artifact. But examples in CAMPRule, coming out of this translation, are available in the directory `~/samples`. In particular, the example of Section 2.1 is in `~/samples/camp_rule/icfp17.camp_rule`. The rest of the compilation chain as well as the entire compiler presented in Figure 3 is available in the directory `~/qcert/coq`. The CAMPRule language and its relationship with JRules are described in the paper *Calculus for Aggregating Matching Patterns* (Shinnar et al. 2015) and thus can serve as a reference to write new examples.



Section 3 of the paper describes specific parts of the compiler. All the corresponding code and proof are available. In this section, we also mention tests. Part of the tests that we wrote are available (`~/samples`), but tests used internally by the ODM developers cannot be public.

Section 4 presents our methodology to translate the Coq compiler into Java. The Java code being part of the product cannot be public. The artifact illustrates the generation of the ASTs as s-expressions which is used to build arbitrary compilation paths between Coq and Java.


Section 5 discuss aspects of the Coq code which is available.

Throughout the paper, there are links to the documentation automatically generated from the Coq code. The name of the HTML file corresponds to the name of the Coq file which is available in `~/qcert/coq`. The name after # character is the name of value defined in the file. For example, the link <https://querycert.github.io/icfp17/doc/NRA#nra> corresponds to the definition of the `nra` in the file `~/qcert/coq/NRA/Lang/NRA.v`.

## 4 CODE ORGANIZATION

The general architecture of the compiler is represented figure 1. It represents all intermediate languages  and the different compilation paths .<sup>2</sup> The languages are:

- $NRA^\lambda$  : NRA with Lambdas

<sup>2</sup>The symbol  provides a link to the corresponding source code.

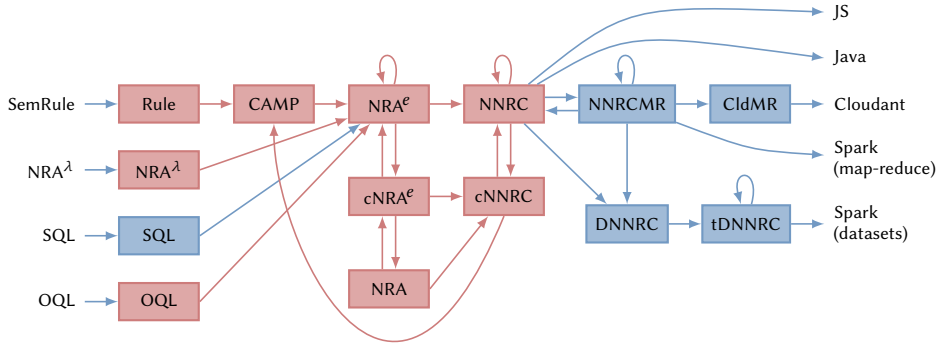


Fig. 1. Compiler architecture. The figure provides hyperlinks to the corresponding parts in the source code.

- SQL : Structured Query Language
- OQL : Object Query Language
- CAMP : Calculus of Aggregating Matching Patterns
- Rule : Rule Macros for CAMP
- NRA : Nested Relational Algebra
- NRA<sup>e</sup> : NRA with Environments
- cNRA<sup>e</sup> : Core NRA<sup>e</sup>
- NNRC : Named Nested Relational Calculus
- cNNRC : Core NNRC
- DNNRC : Distributed NNRC
- tDNNRC : Typed DNNRC
- NNRCMR : NNRC + Map/Reduce
- CldMR : NNRC + Cloudant Map/Reduce

The general organization of the source code available in `~/qcrt` is the following:

- `bin`: directory where binaries are installed
- `camp-java`: CAMP AST in Java
- `coq`: the Coq source code of the compiler
- `javaService`: code to integrate Java frontends into web demos and the `qcrt` command line
- `jrules2CAMP`: Java code allowing the integration of the ODM jar files
- `lib`: directory where libraries are installed
- `ocaml`: OCaml code of the command line compiler
- `runtime`: runtime library for the different backends
- `samples`: examples of queries
- `scripts`: utility scripts
- `sqlParser`: SQL parser
- `sqlppParser`: SQL++ parser
- `www`: JavaScript interface of the compiler

So the most interesting directory is `coq`, the compiler in Coq. In this directory, each language has its own directory with a subdirectory `Lang` containing the AST and the definition of the semantics of the language. Then, depending on the language, some additional subdirectory can be present. For example, it is possible to find a `Optim` subdirectory containing the optimizer for the language,

or a typing subdirectory containing the type checker. The directory Basic directory contains some the data-model which is shared between all languages. This directory also contains some utility functions. The directory Translation contains all the translation functions that compiles a language into another. Finally, the directory compiler contains the compiler driver.

## 5 BUILD THE LINUX IMAGE CONTAINING THE ARTIFACT

The following instructions are not necessary to use the artifact. They are here to document how we built it.

### 5.1 Install Linux

The image is based on Debian 8 "Jessie" (<https://www.debian.org>). The user name is qcirt and the password is querycert.

### 5.2 Install dependencies

Install some basic utilities:

```
sudo apt-get -y install ssh openssh-server emacs-nox vim-nox
```

Install OCaml and Coq:

```
sudo apt-get -y install m4 opam
opam init
eval `opam config env`
opam switch 4.04.1
eval `opam config env`
opam install coq.8.6 ocamlbuild.0.11.0 menhir.20170418 base64.2.1.2
```

Install Java and Scala:

```
apt-get install apt-transport-https ca-certificates
echo "deb http://http.debian.net/debian jessie-backports main" | sudo tee -a /etc/apt/sources.list
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
sudo apt-get -y update
sudo apt-get -y -t jessie-backports install openjdk-8-jdk scala sbt
```

Install Q\*cert:

```
cd
wget https://querycert.github.io/icfp17/resources/icfp17.pdf
wget https://querycert.github.io/icfp17/resources/README.md
wget https://querycert.github.io/icfp17/resources/samples.tgz
tar xvfz samples.tgz
rm samples.tgz
git clone https://github.com/querycert/qcirt.git
git checkout c7e017e2f475f4435cd8d867a43eca7c80471e9e
cd qcirt
make -j 9 qcirt && make extraction && make javacode && make -C samples
echo 'export PATH="$HOME/qcirt/bin:$PATH"' >> ~/.bashrc
cd
```