Artifacts Available acm
Artifacts Evaluated Functional acm

Prototyping a Query Compiler using Coq (Experience Report)

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

Designing and prototyping new features is important in many industrial projects. Functional programming and formal verification tools can prove valuable for that purpose, but lead to challenges when integrating with existing product code or when planning technology transfer.

This article reports on our experience using the Coq proof assistant as a prototyping environment for building a query compiler intended for use in IBM's ODM Insights product. We discuss the pros and cons of using Coq for this purpose and describe our methodology for porting the compiler to Java, as required for product integration.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Information systems** → *Database query processing*;

Additional Key Words and Phrases: query compiler, proof assistant, business rules

ACM Reference Format:
Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2017. Prototyping a Query Compiler using Coq (Experience Report). *Proc. ACM Program. Lang.* 1, ICFP, Article 9 (September 2017), 15 pages.
https://doi.org/10.1145/3110253

1 INTRODUCTION

This paper[1] presents our use of Coq [The Coq development team 2016] to prototype new functionality in a commercial software product: a query compiler from IBM's *Operational Decision Manager: Decision Server Insights* (ODM Insights) [IBM 2017], which uses a rule-based DSL, to the Cloudant [2015] distributed database, which uses map/reduce and JavaScript. The purpose of a proof assistant in this context is not to obtain a certified compiler but to allow proofs to be used as an additional software engineering tool for the development and specification of the compiler.

Two aspects that we identified as particularly challenging and benefiting from the use of formal methods were: (1) the preservation of the source language's semantics despite the large gap between the source language and the target language and (2) the implementation and customization of query optimization techniques, which is always a delicate task. In addition to its theorem proving

[1] An earlier version of this paper was published, in French, in [Auerbach et al. 2017b].

2475-1421/2017/9-ART9
https://doi.org/10.1145/3110253

capabilities, Coq is well suited to writing compilers, being a functional programming language with pattern matching.

Other formal systems combining programming and verification could have been used: we chose Coq primarily for its familiarity. At the beginning of the project, one of the team members had prior expertise in Coq and some other members were OCaml programmers. Previous projects on verification of data management systems [Benzaken et al. 2014; Malecha et al. 2010] offered additional evidence as to the maturity of Coq in that context, and discussions with Véronique Benzaken and Évelyne Contejean provided an initial impetus in this direction.

This project began in 2014 with three people and initially focused on understanding the semantics of the source language and an initial translation to a classical database model [Shinnar et al. 2015]. This effort resulted in a Coq prototype of about 7k lines of specifications and 10k lines of proofs.[2] While this first development sufficed as a specification, it was far from being a credible prototype for a product. Notably, it lacked support for code-generation and necessary optimizations. At that point, the decision was made to continue using Coq to build a full prototype, leveraging this initial development as the basis for the whole project.

Two additional people joined the project in 2015 and Stefan Fehrenbach, a student, did a three month internship on a component disconnected from the parts intended for product delivery. With each team member working in parallel on other projects, we estimate that roughly four person-years were invested in this effort.

The Coq source code today contains around 35k lines of specification and 42k lines of proof. The extracted OCaml code is around 80k lines, supplemented by 8k additional lines of handwritten OCaml code, which includes parsers, pretty-printers and miscellaneous utilities. Integration with ODM Insights consists of several parts: (i) approximately 1.4k lines of Java to translate the ODM Insights internal query representation into the initial intermediate language of our compiler, (ii) approximately 400 lines of Java to integrate the compiler within ODM's designer framework, (iii) approximately 400 lines of JavaScript code and 550 lines of Java code to support query execution in the JavaScript and Java based backends.

The resulting query compiler from ODM Insights to Cloudant written in Coq was translated manually into Java code for delivery to the product team. This translation took one person about three weeks. The resulting compiler contains 11k lines of Java code (including the 1.4k lines of Java already written for the translation of the internal representation used by ODM Insights). This Java code is smaller than the Coq specification for two main reasons. First, the Java translation contains only the definitions of the abstract syntax trees for the compiler's intermediate languages and the translations between these languages, but does not include any evaluation code for those languages. Second, the Coq prototype has grown into a research platform, with additional input and output languages and compilation paths that are not part of the Java translation.

In Section 2, we give an overview of the query compiler and describe its architecture. In Section 3, we describe the role played by proofs, and notably how we used a combination of tests and proofs during the development of the prototype. In Section 4, we present our integration of the compiler with the product and the methodology used to port the prototype to Java. In Section 5, we discuss some of the lessons learned from using a theorem prover for prototyping purposes. We conclude the paper with a short review of the state of the art in Section 6 and with some final remarks on the costs and benefits of using Coq as a software engineering tool in Section 7.

For the interested reader, Q*cert, the research Coq compiler is available at https://querycert.github.io/icfp17. This paper uses the symbol ✿ to link to the Coq code corresponding to the part of the project described.

[2] The number of lines of code is computed using `coqwc` for Coq and `cloc` for the other languages.

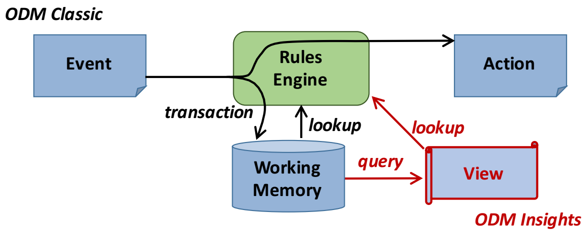

Fig. 1. Managing events in ODM Insights.

2 QUERY COMPILER TO CLOUDANT FOR ODM INSIGHTS

Before discussing the methodology used to develop our prototype, we first present the project's objective and the compiler architecture. This work was done as part of the *Middleware for Events, Transactions, and Analytics* project [Arnold et al. 2016].

2.1 Queries in ODM Insights

The core of ODM Insights is a business rule management system written in the *JRules* family of languages. There are both production rules and queries. Production rules consist of an event given by a `when` clause, a condition given by an `if` clause, and an action given by a `then` clause. The structure of a rule enables a syntax similar to a natural language, which is useful for applications written in collaboration between programmers and business specialists. They allow companies to identify new opportunities (e.g., sales, marketing), risks (e.g., fraud detection, compliance), or to react quickly to certain events (e.g., transactions, delivery notification). Let us consider an example to illustrate this event-condition-action structure. The following rule is executed when an order is received if the account status of the customer is empty and the cost of the order exceeds the average cost of orders. For simplicity, the `then` clause is elided:

```
when a purchase occurs, called LAST
if the customer status of 'the customer' is NONE
   and the total amount of all order items in the order items of LAST
       is more than 'average order amount'
then ...
```

JRules is a modern evolution of production rule languages resulting from work on expert systems in the 70s, especially OPS5 [Forgy 1981]. The rule engine architecture is illustrated in Figure 1. The rule engine maintains a *working memory*, containing a set of *facts* that can be used by the rules. When an event is received (e.g., an order in the previous rule), a fact is added to the working memory and the rules are re-evaluated, taking into account the new state of the memory.

The rules can refer to values calculated globally on the working memory. In the previous example, this is the case for `'average order amount'`. This value is specified by the following rule that calculates the average cost of completed orders containing at least two items.

```
define 'average order amount'
  as the average amount of the order items of all orders,
  where the fulfillment of each order is true
    and the number of elements in the order items of each order is at least 2,
  evaluated every day at 9:28:58 AM
```

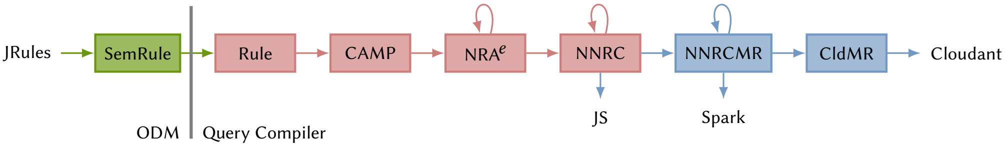

Fig. 2. Compilation path from JRules to Cloudant.

This rule corresponds to a query on the entire working memory. Because the system must respond quickly to events, it is not possible to execute these queries for each event. These are compiled, then materialized as *views* that are updated, either periodically or incrementally. The bottom-right part of Figure 1 shows the part of the system that handles the queries.

The goal of our prototype is to compile this type of query when the working memory is stored in a Cloudant database. At the beginning of the project, ODM Insights only supported the execution of queries over a working memory of Java-like objects maintained in the IBM Websphere eXtreme Scale distributed cache.

2.2 Compiler Architecture

JRules directly manipulates Java objects, while Cloudant uses JSON and expresses requests in the form of map/reduce views containing JavaScript code. To bridge the gap between these two systems, the compiler relies on many intermediate representations. Figure 2 shows the complete compilation path.

Frontend. JRules is a generic term we use to describe a family of languages used in ODM Insights. These different languages are all translated into a common internal language called *SemRule*. Let's illustrate this language by writing the `'average order amount'` rule in the concrete syntax of SemRule:

```
1   query `average order amount` {
2     packageExecutionName = ""
3     ruleExecutionName = "average_order_amount"
4     localQueryResult : aggregate {
5         collectclass1 :
6           Order(collectclass1.fullfillment
7                 && IlrCollectionUtil.getSize(collectclass1.orderItems) >= 2);
8         local0 : OrderItem() in collectclass1.orderItems;
9       }
10      do { avg {Double.valueOf(((double)local0.amount)).doubleValue()}; }
11  }
```

Without trying to explain all the details of the syntax, let us highlight some important aspects. The semantics of business rules, inherited from logic programming, is based on a notion of unification between language expressions and objects in working memory. In this example, lines 5 to 7 use the notation '`var : ClassName(cond)`' which means: if an object in the working memory is a subtype of the class `ClassName` and satisfies the condition `cond`, then instantiate the variable `var` with the corresponding object. As can be seen in the expression on line 8, unification can use a variable defined previously in the rule (`collectclass1`) and access a field (`orderItems`) of the object it is bound to.

Aggregate queries are written as rules using an extension that allows unification on *the set of facts* in working memory rather than each individual fact. In our example, this is the expression `aggregate {...} do {...}` which span line 4 to line 10. The `aggregate` clause accumulates all combinations of objects that unify with the given conditions, and returns a set of instantiated variables. The `do` clause contains an expression that can access these variables, and an aggregate function, in this example `avg`. The expression `aggregate` is a form of comprehension [Tannen et al. 1992; Trinder and Wadler 1988].

In order to capture the semantics of JRules in Coq, we defined the CAMPRule language ✿ and its core CAMP ✿ (Calculus for Aggregating Matching Patterns) [Shinnar et al. 2015]. CAMP relies on pattern matching and supports the unification of Java objects and aggregation expressions as described above. Details about this support for objects are presented in [Shinnar and Siméon 2016].

Middle-end. For the core of the compiler, we decided to rely on well-known representations in databases. Developing a query optimizer is a non-trivial task, in terms of both development time and architecture, as well as in terms of defining the set of necessary optimizations. In this context, existing research in the database literature (see for example Graefe [1993]; Moerkotte [2014]; Ullman and Widom [2000]) guided the choice of intermediate representations.

We therefore chose to translate CAMP to NRA, the Nested Relational Algebra ✿ [Cluet and Moerkotte 1993]. This combinator-based representation eliminates variables and simplifies optimization. Nevertheless, queries coming from CAMP proved difficult to optimize. We therefore introduced NRA^e ✿, a conservative extension of NRA with an environment, which simplifies optimization [Auerbach et al. 2017a].

Backend. In order to produce simple imperative code, we translate NRA^e to NNRC, the Named Nested-Relational Calculus ✿ [Van den Bussche and Vansummeren 2007]. NNRC corresponds to a small functional language (first order) with list comprehension for which it is easy to produce JavaScript ✿.

NNRC also serves as a basis for introducing the map-reduce model. NNRC Map-Reduce (NNRCMR) ✿ is a language where all computations are expressed as *map* and *reduce* functions defined themselves in NNRC. At this stage of compilation, we now have a notion of distributed computation with map/reduce functions containing code that we know how to translate into JavaScript. This intermediate language can be translated directly to the map-reduce subset of Spark [Zaharia et al. 2012] ✿.

The last intermediate language is CldMR ✿. This is also a map-reduce model where functions are written in NNRC, but this time the model takes into account the specifics of Cloudant. For example, the reduce functions in Cloudant are more constrained in that they must be callable an arbitrary number of times to enable the incremental construction of the result. There are also some predefined reduce functions (`sum`, `count`, `stats`).

From CldMR, the translation to Cloudant is direct. The map-reduce structure is the one expected by Cloudant. To generate the map-reduce functions' code, we reuse the compiler from NNRC to JavaScript.

Handling other languages. For research purposes, we have also used the compiler to experiment with other languages. The compiler supports as input subsets of SQL and OQL [Berler et al. 2000] and can produce code for Spark Datasets. Figure 3 depicts the set of languages and compilation paths.

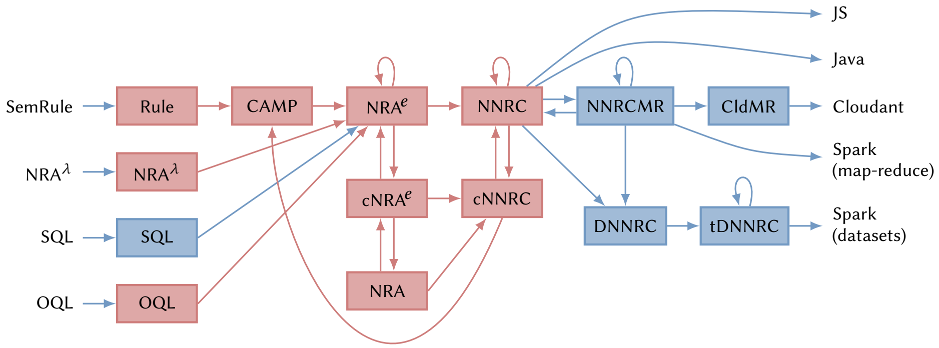

Fig. 3. The complete compiler in Coq. Each box and arrow is a link to the corresponding part in the code.

3 PROOF AND TESTS

The desired correctness result for the compiler is that, for each SemRule query for which Cloudant code is generated, the result of the evaluation of the query on Cloudant is the same as the result of the evaluation with the current ODM Insights implementation.

It is not realistic to formally verify this property in Coq since the semantics of ODM Insights are given by its Java implementation and the semantics of Cloudant by its Erlang implementation. We therefore rely on a mix of proofs and tests to validate the correctness of the compiler.

In the rest of this section, we describe for each part of the compiler the techniques that were used to validate it. The colors in Figure 2 distinguish the proven parts in red from the unproven parts in blue.

Intermediate Languages. Let us first illustrate how languages are defined. To do this, let us take the example of NRA^e. First, we define the abstract syntax of the language using an inductive type: ✿

```
Inductive nraenv_core : Set :=
| ANID : nraenv_core
| ANConst : data -> nraenv_core
| ANMap : nraenv_core -> nraenv_core -> nraenv_core
...
```

Then, the semantics is defined as an evaluation function: ✿

```
Fixpoint nraenv_core_eval (op:nraenv_core) (env: data) (x:data) : option data :=
  match op with
  | ANID => Some x
  | ANConst rd => Some (normalize_data h rd)
  | ANMap op1 op2 =>
    let aux_map d :=
      lift_oncoll (fun c1 => lift dcoll (rmap (nraenv_core_eval op1 env) c1)) d
    in olift aux_map (nraenv_core_eval op2 env x)
  ...
  end.
```

This function is explicitly provided a query op, an environment env, and input data x. It is also implicitly parameterized by a type hierarchy h and a constant environment c used to model the working memory.

The evaluation functions serve as specifications for the proofs, and are also used to run tests.

Frontend. SemRule having no formal semantics, the frontend is checked by means of tests. The result of the evaluation obtained by the CAMPRule interpreter (evaluation function) is compared to that obtained via ODM Insights. We employed a test suite used internally by the ODM developers.

One important task of the test infrastructure is handling the input data and the expected result. They are imported into ODM, assigned types, and exported to a JSON format that includes type annotations. The CAMPRule interpreter uses this JSON to compare the two results. Since the translation from CAMPRule to CAMP has been formally proven, tests at the level of the CAMP language are sufficient to debug the initial translation of SemRules.

To this end, we developed an instrumented version of the CAMP interpreter which, in addition to evaluating the program, retains information about the execution environment. When an error is detected, this interpreter returns an annotated version of the input program that accurately identifies the location and reason for the problem (mainly type errors). This version of the interpreter is proved equivalent to the original version (which defines the semantics) modulo the additional information in case of error. This proof prevents the debugger from diverging from the specification as the language evolves. This interpreter was useful for the debugging of the frontend when it produced incorrect CAMP code.

Middle-end. The core of the compiler is written entirely in Coq and therefore does not have to address interoperability with external tools and formats like the frontend. The part that goes from CAMP to optimized NNRC is fully proven to be semantics preserving.

To illustrate the subtleties that motivated us to formally verify the middle-end, consider two classical equivalences of the relational algebra [Ullman and Widom 2000]. These equivalences are used in the NRA^e optimizer (σ and $\cup$ are selection and set union, respectively):

$$\sigma_{p_1 \wedge p_2}(q) \equiv \sigma_{p_1}(\sigma_{p_2}(q))$$
$$\sigma_{p_1 \vee p_2}(q) \equiv \sigma_{p_1}(q) \cup \sigma_{p_2}(q)$$

Despite their simplicity, neither of these two equivalences are true! More precisely, they are true only under certain assumptions that must be guaranteed by the optimizer. The first equivalence is true only for a well-typed query, that is, in the absence of errors during execution. The second is true for set union, but in the case of multi-sets,[3] it is true only for the maximal union [Libkin and Wong 1994]. Consider, for example, the application of the first relational equivalence in a wrongly typed example (here *age* contains a number, not a string):

$$\sigma_{age=\text{"}John\text{"} \wedge false}(q) \not\equiv \sigma_{age=\text{"}John\text{"}}(\sigma_{false}(q))$$

The evaluation of the expression on the left fails, but the right one returns the empty set. This simple example shows that it is relatively easy to forget key preconditions when implementing a rewrite rule. Note that the corresponding chapter in Ullman and Widom [2000] does not mention aspects related to typing.

The NRA^e optimizer is composed of more than seventy of these kinds of rewrites. Proving them is usually not difficult. The proofs put forward subtle preconditions and also capture the errors of inattention that arrive easily in this type of code. To prove translation correctness between two languages, we use the evaluation functions that define their semantics. Let us illustrate this with the translation from CAMP to NRA^e. The semantics of CAMP is defined by the function `camp_eval` ✿ and the semantics of NRA^e is defined by the function `nraenv_core_eval` ✿. The translation function from CAMP to NRA^e is `nraenv_core_of_camp` ✿. From these three functions, we can state the correctness property of the translation: ✿

[3] The semantics of SQL and most of the query languages used in practice, including JRules, rely on a multi-set.

```
Lemma nraenv_core_of_camp_correct h c q env d:
  lift_failure (camp_eval h c q env d) =
    nraenv_core_eval h c (nraenv_core_of_camp q) (drec env) d.
```

This lemma is parameterized by the type hierarchy h, a constant environment c corresponding to the working memory, the input query q, the execution environment env and input data d. It illustrates that there is a different encoding of the environment in the two semantics: env (an associative list) becomes drec env (a record). Similarly, the function lift_failure translates the error encoding between in CAMP in NRA^e.

Even if this part is fully proved, it is possible that the code does not behave as intended due to erroneous or incomplete specifications. As the specifications are given by the interpreters, it was important to debug them. Proving correctness of the translation is an effective debugging tool which forces looking at the specifications in detail. Moreover, as we have proved the correctness of the translation with CAMP and the semantics of CAMP have been tested, it means that the semantics of the following languages are also correct with respect to the tests.

For the complexity of development in Coq, several points are to be noted. Among the most demanding proofs are those devoted to two underlying aspects of the infrastructure: the type system ✿ and renaming and substitution for NNRC ✿. The type system itself is relatively complex, including two notions of records (open/closed), and a hierarchy of classes [Shinnar and Siméon 2016]. It is also essential to the rest of the compiler because the absence of typing errors is a precondition in most rewrites for NRA^e and NNRC. Additionally, the proof that optimizations for NRA are also valid for NRA^e ✿[4] required significant effort. In the rest of the code, the difficulty has more often been choosing the right formulation of the properties to prove than the actual proof. An important example in this category is the definition of equivalence between two expressions for NRA^e ✿ and NNRC ✿.

Backend. The correctness of translations from NNRC to CldMR is verified by a mix of tests and proofs. Eventually, we hope to fully prove these translations. At the moment, some important properties of the translation from NNRC to NNRCMR have been proven and individual optimizations on NNRCMR are proven correct, but their composition has not been proven correct.

The correctness of code generators to JavaScript and Cloudant is based on testing. Their debugging was mainly done using traditional tools and techniques from the target languages. For example, in JavaScript it is easy to add print statements. Debugging Cloudant code, a distributed system running on the cloud, is a much more difficult task. To mitigate this difficulty, we implemented a naive backend from NNRCMR to Spark (but keeping the map and reduce functions in JavaScript). The resulting Spark code is simpler to debug than the corresponding Cloudant code.

It should be noted that even though this part used Coq primarily as a simple functional language, the proof aspects were still useful. Proving properties was an effective way to find bugs and verify assumptions and invariants. However, sometimes the need to maintain proofs was a burden. Since this part of the compiler was less mature than the rest of the project, code changed frequently. Maintaining the proofs could be tedious and so sometimes we simply deleted them.

4 INTEGRATION INTO THE PRODUCT

Integration with the product was done in several steps. The first steps employed a machine translation of the Coq code. In the final steps we hand-translated the compiler into maintainable Java source.

[4] Theorem 1: Equivalence Lifting in [Auerbach et al. 2017a].

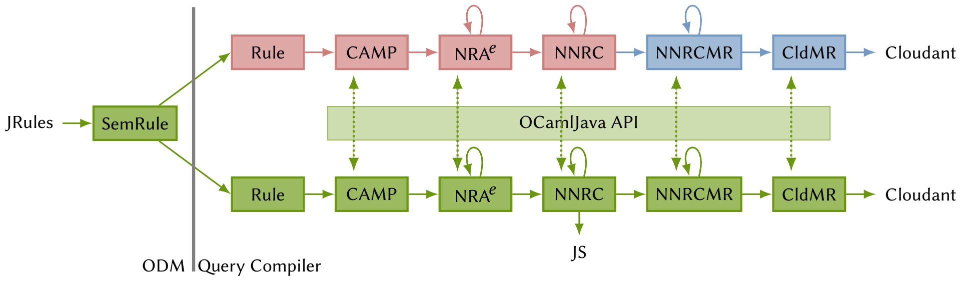

Fig. 4. Dual Compilation Chain in OCaml and Java.

4.1 Using Coq with ODM Insights

In Figure 2, the part on the left of the vertical bar is written in Java and is part of ODM Insights. The part to the right of the bar is written in Coq and is part of our prototype. In order to use our Coq compiler on queries coming from ODM Insights, we had to program the arrow crossing the border. That part was also written in Java.

We used three different techniques to communicate between Java and Coq. Each technique was used for a different purpose. The first technique translated SemRules to Coq source files. Each such file contained the query directly as a Coq value of type `camp_rule` ✿. This technique was particularly useful at the beginning of the development process. It enabled us to evaluate examples in the interactive mode of Coq, simplifying unitary tests of functions.

The second communication technique translated SemRules to text files, using a concrete syntax for CAMPRule. The concrete CAMPRule syntax can be considered an exchange format. To use this technique, the Coq compiler is extracted to OCaml and is combined with a parser written with Menhir [Pottier and Régis-Giannas 2016]. This technique produces a command line compiler that takes inputs written in CAMPRule. It provides a simple and efficient way to execute regression tests on queries coming from ODM Insights.

Finally, the third method of communication between Java and Coq is to use OCamlJava [Clerc 2015], a compiler from OCaml to Java bytecode. From the Coq code extracted in OCaml, we defined a library that can be compiled with OCamlJava, enabling the invocation of the compiler directly from Java. This method was used to embed the Coq compiler directly into the product.

4.2 Re-Implementing the Compiler in Java

We have just seen that the Coq compiler can be embedded in ODM Insights via mechanical translation. However, as the ODM Insights code is primarily written in Java, and its development team is most familiar with that language, providing the code as Coq source was not realistic. Even if an extraction from Coq to Java source were available, we would expect the result to be only marginally more understandable than the Java bytecodes produced by OCamlJava. To produce clear Java code that can be maintained by the team, a manual translation was required. To avoid the addition of a new language, we didn't use Scala [Odersky et al. 2016] even though it is a JVM based language and the translation would have probably been closer to the Coq code.

The translation from Coq to Java raises two problems: (1) the proofs of correctness made in Coq are not preserved by the translation and new errors can easily be introduced; (2) once the Java code is entrusted to the development team, and the Coq code is left with the research team, both versions are likely to diverge. None of these problems have an ideal solution, but we have put techniques in place to try to overcome them.

Ensuring an Accurate Re-implementation. In order to carry out the translation of the Coq code to Java, we created an infrastructure enabling progressive translation of the code with correctness testing of the translation at each step. This infrastructure is presented in Figure 4.

The compilation path at the top of Figure 4 follows the path from Figure 2 and is the original implementation in Coq. The compilation path at the bottom of Figure 4 represents the manual re-implementation in Java. The abstract syntax trees of each language are implemented as Java class hierarchies that mimic the structure of the Coq algebraic types.

The dashed vertical lines represent the ability, at each compilation step, to switch between Coq and Java representations. The internal representation of each abstract syntax tree is different but isomorphic and shares the same semantics. Communication between the two representations is done by encoding the trees as S-expressions. It is thus possible to take arbitrary paths through this dual compilation chain.

There are two key advantages to this dual compilation chain. First, each step in Java can be created and debugged independently. Second, a given step can be executed in both Java and Coq and the abstract syntax tree structures produced can be compared. In an end-to-end test, when a single step is in Java and the others remain in Coq, it is easy to attribute the error to the code in Java. Also, if several steps had been tested simultaneously, several errors could compensate for each other and hide translation errors.

Moreover, the structural comparison of the ASTs after each step provides strong confidence in the fidelity of the behavior of the Java code with respect to the Coq code. It can find errors that produce correct code but not necessarily identical code (e.g., missing optimizations). A textual comparison of the ASTs on the S-expressions would have been too restrictive since different variable names may be chosen depending on the compilation path.

The reuse of ODM Insights tests with our dual compilation chain gives us great confidence in the fidelity of our translation from Coq to Java. During the translation of the optimizer, when we tried to debug the Java version of our compiler, we soon realized that we did not have a good intuition for what optimizations were being applied and in what order. To solve this problem, we instrumented the optimizers to trace their steps. A Coq optimization trace can be compared with the equivalent Java optimization trace to confirm that the same optimizations are applied in the same order.

Correlating Coq code with idiomatic Java code. Some independent evolution of both the Coq and Java code is inevitable so we have structured the Java code to make it easy to compare manually against the original Coq code. Another goal is to write idiomatic Java code that can be easily understood and adopted by the product team. This goal can sometimes conflict with the goal of writing Java code that can be compared with the Coq version.

In order to ease the comparison between the Coq and the Java code, we keep the same modular structure, down to the individual function level. Moreover, each function translated into Java contains the corresponding Coq code in the comment header. We illustrate this with the rewrite presented in Section 3 (other comments, dedicated to Java programmers, are elided):

```
/**
  From TOptimEnvFunc.v: last checked 5/2/2016
  Definition tselect_and_fun {fruntime:foreign_runtime} (p: algenv) :=
    match p with
    | ANSelect op1 (ANSelect op2 op) => ANSelect (ANBinop AAnd op2 op1) op
    | _ => p
    end.
*/
```

```
private static class tselect_and_fun implements OptFun {
  @Override
  public NraNode optimize(NraNode nra) { ... }
}
```

If the Coq function is long, the header can be shortened and the rest of the Coq code is then added as comments in the body of the function next to the corresponding Java code. This allows those programmers who understand Coq to see the intent of the translation and also to check if an update is needed. The `last checked` information in the header enables finding in the Coq code repository the changes made since the last synchronization between the Java code and the Coq code. In the Coq code, the link to the Java code is also provided in a comment.

During the re-implementation, we had to solve some challenges like dealing with higher order functions. We wanted to keep the code compatible with Java 7, and so did not use Java lambdas. Therefore, we used classes, interfaces, and subtyping to implement a sufficient equivalent for those thing that Coq did with higher order functions.

Other challenges came from the fact that Java is an imperative language for which library functions are rarely purely functional. For example, sorting an array is done in place and merging two dictionaries can modify one of them. For all these functions, we have implemented a wrapper that introduces a copy in order to give a consistent functional semantics to these operations. It should be noted that the imperative nature of Java also has advantages. For example, in Coq, the generation of fresh names requires considerable effort whereas in Java it is simple, since a global counter can be employed.

For business reason, the product team have not yet taken the ownership of the prototype because they have to work on priority features. Therefore, we do not have feedback on the effectiveness of the methodology put in place to keep the Coq and Java code in sync.

5 DISCUSSION

In an industrial context, the use of a theorem prover such as Coq changes the development experience considerably. We report here on the aspects that required the most attention from the team or had consequences that were not immediately obvious at the start of the project.

5.1 Impact of Early Design Choices

Executable Semantics and Extensible Data Model. One of the important, and early, design choice is the use of executable semantics for the various intermediate languages. As we have seen in Section 3, one rationale for that choice is the consequent ability to combine proofs with actual tests against evaluation in ODM or Cloudant.

This semantics relies on the presence of a full fledged data model that can be extracted to OCaml. The first Coq prototype used a data model with simple primitive types (integers, Booleans and strings) which were only a subset of the types supported in the product (notably including floating point numbers and dates). In most other programming languages, handling those would be a matter of identifying appropriate libraries. In the Coq context, the standard library handles only a limited set of primitive types, and designing additional ones (e.g., for dates) can represent a large effort which was not realistic within our time-frame.

Use of Type Classes. To solve this problem, we made our compiler extensible. The goal is to enable the addition of new primitive types without requiring systematic revision of all the proofs. For this we use type classes [Sozeau and Oury 2008]. In practice, this means that the compiler's code is parameterized by multiple type classes that give an axiomatization for the extension points required for the external primitive types and their operations.

The parameterization of the compiler through type classes is split into two main parts: a set of type classes for the definition and runtime behavior of foreign primitive types, and another set of type classes for type-checking operations on those types. While much of the code only assumes one of those two bundles, they notably intersect in the query optimizers which assume type-correctness for much of their logic, and in the overall compiler driver. To simplify extraction (as the extracted code must explicitly provide instances of the required type classes at each function call), we have created functors that internally instantiate the type classes.

While this solution solves the problem, it adds complexity to the entire code, and this part of the development has sometimes been a source of frustration for the team. The difficulty in identifying which subset of the type classes are necessary within a particular part of the code is often compounded by the complexity of the error messages produced by the Coq type system when using them. While a systematic use of functors throughout the compiler may have been a better alternative, type classes initially appeared as a lighter weight solution. More experiences and some guidance on those questions from the Coq community would certainly prove valuable.

5.2 Tooling

Code navigation and notations. Tooling is another area of notice. An obvious observation is that Coq tools heavily emphasize support for theorem proving rather than traditional software engineering tasks. We used Proof General throughout the project, with one team member using the Coqoon plugin for Eclipse. As more team members came on board, the lack of code navigation support was at times a nuisance, a difficulty sometimes compounded by the use of notations. With Coq supporting notations that include unicode symbols and many of our intermediate representations (such as NRA or NNRC) derived from languages existing in the literature, it seemed natural to reuse classical notations as much as possible (e.g., σ for the selection operation and $\cup$ for the union). As some of these notations are available in several languages of our compiler, this has also led to the use of Coq's notations scopes to specify which language is intended. The use of these notations led to disagreement within the team. While it is nice that Coq theorems look like their formulations in articles, it is sometimes necessary to use the ASCII version of the names inside proofs. This forces us to manipulate a double set of names.

Refactoring. All of those issues were additional incentives for improving the code's organization and naming conventions, and led to several rounds of refactoring. Those turned out to be much more labor intensive than expected, and are still ongoing. One of the reasons for the amount of work involved in refactoring is the necessity to maintain proofs. In addition to the size of the proofs, they sometimes rely on tactics that may or may not be robust to changes in the structure or names used in the rest of the code. Identifying the interdependency between proofs and code can be difficult and a large amount of person-hours were spent on those efforts as a result. In more traditional software development contexts, numerous tools are available for renaming, and moving code, and identifying dependencies. Having access to some level of refactoring functionality in Coq would have been welcome and have most likely saved the team a lot of tedious work.

5.3 Proofs for Software Engineering Purposes

We did not use proofs only to guarantee semantics properties. We also used proofs to help maintain the compiler. A striking example is the compiler driver.

Figure 3 depicts many possible paths through the compiler (including cyclic paths). These paths are represented in Coq by the `driver` ✿ data structure. We used Coq to prove some properties of this driver.

For example, the function `get_path_from_source_target` ✿ computes a default compilation path between two given languages. This function is used as part of the command line driver of the compiler, allowing the user to specify a source and target language without needing to specify all the intermediate steps. This function has 1831 lines of code. It is easy to make a mistake or forget a path. To ameliorate this problem we have proved the property that for all drivers, if we extract the source and target languages of the driver, then the function `get_path_from_source_target` will return a valid compilation path between these two languages ✿.

This kind of property improves our confidence in the code and also simplified maintenance. In particular, when we add a new language, it forces (and guides) us to update all the corresponding parts of the driver.

6 RELATED WORK

The considerable progress made by proof assistants, such as Coq and Isabelle, over the last ten years is well known. Several projects show that it is possible to use these tools for the development of large systems, whether the goal is for formal mathematics [Gonthier et al. 2013] or for software certification [Fisher 2014; Klein et al. 2009; Leroy 2012]. Our work is placed in a context closer to software engineering, using Coq as one tool among others for development and verification, along the line of Jorge et al. [2007]. We found few similar projects in the literature, suggesting that this is still marginally used in industry.

Having chosen Coq to develop the compiler prototype forced us to translate the code into Java. Translating and then maintaining the two compilers has a non-negligible cost. An alternative that would have been interesting to explore is the use of tools that allow us to express properties directly on Java programs and to prove them [Burdy et al. 2005; Marché 2015].

The domain of databases in general, and that of query languages in particular, are characterized by strong links between theory and practice. Since the work of Trinder and Wadler [1988] in the 1980s, the formalization of query languages often relies on functional bases [Fegaras and Maier 2000; Tannen et al. 1992; Van den Bussche and Vansummeren 2007], making them particularly suitable for mechanization in Coq. A number of recent works deal with different aspects of the formalization of database systems (e.g., relational [Benzaken et al. 2014; Malecha et al. 2010], XML [Cheney and Urban 2011], or Linq [Malecha and Wisnesky 2015]). This work is technically complementary to the development presented in this paper.

The development of formal tools and methods for the specification and implementation of query compilers is an important subject in the relational context [Pirahesh et al. 1992]. The complexity associated with the development of these compilers and their use in new application contexts makes the subject remain current [Klonatos et al. 2014]. Our approach is similar to that of Coko-Kola [Cherniack and Zdonik 1996], a query optimizer developed in the 1990s with the Larch prover of MIT.

7 CONCLUSION

The use of Coq for prototyping has been interesting on several levels. Team members with varying levels of prior experience with Coq added a number of unknowns in terms of feasibility and development time. The decision to use Coq not only as a specification tool but also to develop a complete prototype of the compiler created a set of additional challenges. This article provides an overview of our experience as well as the challenges we encountered, and we hope that some of the techniques described may be useful in other contexts.

A posteriori, it is difficult to determine if the additional development effort was worthwhile for this project. The use of a tool like Coq for development requires a significant investment, especially at the start of the project, and represents a significant additional cost (Figure 5 summarize the sizes

AST and semantics:	Lines of Coq Code	Lines of Proof	Lines of Java Code
CAMPRule	0.1k	0k	1.1k
CAMP	0.8k	0.3k	0.9k
cNRAEnv + NRAEnv	2.7k	3.8k	0.5k
cNNRC + NNRC	1.5k	4k	1k
NNRCMR	0.4k	0.7k	0.7k
CldMR	0.5k	0k	0.7k
Optimizers:			
NRAEnv	2.8k	4.5k	3k
NNRC	1.1k	1.7k	1.3k
NNRCMR	0.6k	0.6k	1.5k
from CAMPRule to CldMR:	1k	1k	1.6k

Fig. 5. Summary of the Coq and Java compilers sizes (the Java code doesn't implement the semantics).

of the different parts of the Coq and Java compilers). On the other hand, relying on a fully proven core changes the development experience in some positive ways. When a bug is identified, the search for its origin can immediately exclude a significant part of the code. Coq also encourages the writing of reusable code. In our case, the compiler is built around small, well-defined languages. This investment has notably simplified the addition of other query languages (e.g. OQL and SQL). Based on this experience, we now believe that the realization of a fully certified query compiler is also quite feasible.

REFERENCES

M. Arnold, D. Grove, B. Herta, M. Hind, M. Hirzel, A. Iyengar, L. Mandel, V. Saraswat, A. Shinnar, J. Siméon, M. Takeuchi, O. Tardieu, and W. Zhang. 2016. META: Middleware for Events, Transactions, and Analytics. *IBM R&D* 60, 2–3 (2016), 15:1–15:10.

Joshua Auerbach, Martin Hirzel, Louis Mandel, Avi Shinnar, and Jérôme Siméon. 2017a. Handling Environments in a Verified Query Compiler. In *SIGMOD*.

Joshua Auerbach, Martin Hirzel, Louis Mandel, Avi Shinnar, and Jérôme Siméon. 2017b. Prototyper un compilateur de requêtes avec Coq. In *Vingt-septièmes Journées Francophones des Langages Applicatifs*. Gourette, France.

V. Benzaken, E. Contejean, and S. Dumbrava. 2014. A Coq Formalization of the Relational Data Model. In *ESOP*.

M. Berler, R. Cattell, and D. Barry. 2000. *The object data standard: ODMG 3.0.* Morgan Kaufman.

L. Burdy, Y Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, R. Leino, and E. Poll. 2005. An overview of JML tools and applications. *STTT* 7, 3 (2005).

J. Cheney and Ch. Urban. 2011. Mechanizing the Metatheory of mini-XQuery. In *CPP*.

M. Cherniack and S. Zdonik. 1996. Rule Languages and Internal Algebras for Rule-Based Optimizers. In *SIGMOD*. http://www.cs.brandeis.edu/~cokokola.

X. Clerc. 2015. OCaml-Java. (2015). http://ocamljava.org.

Cloudant 2015. Anatomy of the Cloudant DBaaS. (2015). https://cloudant.com/CloudantTechnicalOverview.pdf.

S. Cluet and G. Moerkotte. 1993. Nested Queries in Object Bases. In *DBPL*.

L. Fegaras and D. Maier. 2000. Optimizing object queries using an effective calculus. *TODS* 25, 4 (2000).

K. Fisher. 2014. Using formal methods to enable more secure vehicles: DARPA's HACMS program. *ACM SIGPLAN Notices* 49, 9 (2014), 1–1.

Ch. Forgy. 1981. *OPS5 user's manual.* Technical Report 2397. CMU.

G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. O. Biha, et al. 2013. A machine-checked proof of the odd order theorem. In *ITP*.

G. Graefe. 1993. Query evaluation techniques for large databases. *CSUR* 25, 2 (1993).

IBM. 2017. Operational Decision Manager. http://www.ibm.com/software/products/en/odm/. (2017).

J. S. Jorge, V. M. Gulías, and D. Cabrero. 2007. Certifying properties of programs using theorem provers. In *Verification, Validation and Testing in Software Engineering*. IGI Global, Chapter 10, 220–267.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, Ph. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *SOSP*.

Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. 2014. Building Efficient Query Engines in a High-Level Language. (2014), 853–864.

X. Leroy. 2012. Mechanized Semantics for Compiler Verification. In *CPP*.

L. Libkin and L. Wong. 1994. Some properties of query languages for bags. In *DBPL*.

G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. 2010. Toward a verified relational database management system. In *POPL*.

G. Malecha and R. Wisnesky. 2015. Using Dependent Types and Tactics to Enable Semantic Optimization of Language-integrated Queries. In *Proceedings of the 15th Symposium on Database Programming Languages (DBPL 2015)*. 49–58.

C. Marché. 2015. *The Krakatoa Verification Tool for Java programs.* INRIA.

The Coq development team. 2016. *The Coq proof assistant reference manual.* LogiCal Project. http://coq.inria.fr Version 8.5pl2.

G. Moerkotte. 2014. *Building Query Compilers.* Univ. Mannheim. http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

Martin Odersky, Lex Spoon, and Bill Venners. 2016. *Programming in Scala: Updated for Scala 2.12* (3rd ed.). Artima Incorporation, USA.

H. Pirahesh, J. M. Hellerstein, and W. Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*. 39–48.

F. Pottier and Y. Régis-Giannas. 2016. Menhir Reference Manual. (2016).

Avraham Shinnar and Jérôme Siméon. 2016. A Branding Strategy for Business Types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer International Publishing, Cham, 367–387.

A. Shinnar, J. Siméon, and M. Hirzel. 2015. A Pattern Calculus for Rule Languages: Expressiveness, Compilation, and Mechanization. In *ECOOP*.

Matthieu Sozeau and Nicolas Oury. 2008. *First-Class Type Classes.* Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

V. Tannen, P. Buneman, and L. Wong. 1992. Naturally Embedded Query Languages. In *ICDT*.

P. W. Trinder and P. L. Wadler. 1988. List Comprehensions and the Relational Calculus. In *GalFp*.

J. Ullman and J. Widom. 2000. *Database Systems: The Complete Book.* Prentice Hall.

J. Van den Bussche and S. Vansummeren. 2007. Polymorphic type inference for the named nested relational calculus. *Transactions on Computational Logic (TOCL)* 9, 1 (2007).

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica. 2012. Fast and interactive analytics over Hadoop data with Spark. *USENIX Login* 37, 4 (2012), 45–51.