Handling Environments in a Nested Relational Algebra with Combinators and an Implementation in a Verified Query Compiler

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

ABSTRACT

Algebras based on combinators, i.e., variable-free, have been proposed as a better representation for query compilation and optimization. A key benefit of combinators is that they avoid the need to handle variable shadowing or accidental capture during rewrites. This simplifies both the optimizer specification and its correctness analysis, but the environment from the source language has to be reified as records, which can lead to more complex query plans.

This paper proposes NRA^e, an extension of a combinators-based nested relational algebra (NRA) with built-in support for environments. We show that it can naturally encode an equivalent NRA with lambda terms and that all optimizations on NRA carry over to NRA^e. This extension provides an elegant way to represent views in query plans, and can radically simplify compilation and optimization for source languages with rich environment manipulations.

We have specified a query compiler using the Coq proof assistant with NRA^e at its heart. Most of the compiler, including the query optimizer, is accompanied by a (machine-checked) correctness proof. The implementation is automatically extracted from the specification, resulting in a query compiler with a verified core.

1. INTRODUCTION

Some recent development around query languages and query processing is happening outside traditional database management systems, e.g., language-integrated queries [15, 28], large-scale distributed processing infrastructure [3, 7, 29], NoSQL databases [30], or domain specific languages [34]. Understanding and guaranteeing correctness properties for those new data processing capabilities can be important when dealing with business critical or personal data. In relational systems, rule-based optimizers and optimizer generators [10, 12, 13, 18, 24, 31] contribute to the high levels of performance and correctness confidence by enabling the specification, verification, and implementation of query compilers. This paper proposes to leverage modern theorem proving technology as a foundation for building well-specified and formally verified query compilers.

Although our motivation stems from new query compilation scenarios, and specifically the extension of a rule-based language with a query DSL, we believe the approach can be applied in more traditional database contexts. As was shown in compilers for language integrated queries [22], relying on traditional database algebras can bring numerous benefits. We follow a similar strategy and build on top of the nested relational algebra (NRA) [14, 17] which has been successfully used for building query compilers for nested data models, notably for OQL and XQuery [27, 32].

As observed in prior work [12, 13], ensuring correctness remains hard even with a rule-based approach, and we have encountered similar challenges. Three of the main challenges are (i) reasoning about scoping when variables are involved as part of the optimization rules, (ii) providing tools to facilitate reasoning and correctness checking, and (iii) handling code fragments as part of the rules. Although most of the previously proposed techniques and optimizations for NRA directly apply, those three challenges require special care and are the focus of this paper.

Handling Environments. The first challenge is intrinsically difficult for any compiler[1] and is the central focus of the paper. Combinator-based algebras [12, 34] have been used to eliminate variables in an attempt to facilitate reasoning. However, they force the query translator to reify environments as part of the data being processed, which can result in larger and more complex query plans.

This paper proposes NRA^e, an extension to a combinator-based nested relational algebra with native support for environments. It avoids blow-ups in query plan size while facilitating correctness reasoning. This extension is conservative in the sense that existing NRA optimizations can be applied even to query plans containing the new constructs.

A Verified Query Compiler. To address the second challenge, we are developing a query compiler using the Coq proof assistant [16] which we use for both the compiler spec-

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

DOI: http://dx.doi.org/10.1145/3035918.3035961

[1] Proper handling of variable scoping is known as one of the main difficulties in solving the POPLmark challenge [5].

ification and correctness proofs. A Coq feature called *extraction* [23] can then be used to automatically generate the query compiler's code from that specification. Let us first illustrate the use of Coq for the implementation and verification of a simple algebraic rewrite. Throughout the paper, we will use the flower symbol ✿ to provide hyperlinks to the corresponding source code.

As in rule-based optimizer generators, optimizations can be written in Coq as rewrites on algebraic terms. As an example, here is the Coq code for pushing down a selection operator over a union operator (based on the classic distributivity law: $\sigma_{\langle q_0 \rangle}(q_1 \cup q_2) \equiv \sigma_{\langle q_0 \rangle}(q_1) \cup \sigma_{\langle q_0 \rangle}(q_2)$):

```
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.
```

This code is written in a functional style and defines a function with name `select_union_distr_fun` and one parameter `q` (the algebraic plan). The body of the function uses pattern matching to check whether the terms in `q` are indeed a selection over an union, in which case it applies the rewrite, or not, in which case it leaves the plan unchanged. A key difference with most rule-based optimizer generators, is that Coq allows the query compiler developer to state (and also prove) the correctness of this rewrite:

```
Proposition select_union_distr_fun_correctness q:          ✿
  select_union_distr_fun q ⇒ q.
Proof.
  tprove_correctness p.
  apply tselect_union_distr.
Qed.
```

This proposition states that for all query plans `q`, applying the function `tselect_union_distr_fun` returns an equivalent query. Here the symbol ⇒ denotes a notion of type-preserving equivalence which is used throughout our optimizer and is formally defined later in the paper. The proposition statement is followed by a proof script that is mechanically checked. The proof relies on an automated proof tactic `tprove_correctness` which eliminates all trivial cases except the important one which is solved using the lemma `tselect_union_distr`. That lemma itself is simply a type preserving variant of the distributivity law for selection over union which we saw earlier:

```
Lemma tselect_union_distr q₀ q₁ q₂ :                       ✿
  σ⟨ q₀ ⟩(q₁ ∪ q₂) ⇒ σ⟨ q₀ ⟩(q₁) ∪ σ⟨ q₀ ⟩(q₂).
Proof. ... Qed.
```

Formal verification techniques have been applied to the formalization of relational [6, 26] and non-relational [11, 34] query languages, but have seldom been used with query compilers implementation in mind. A notable exception is the Coko-Kola project [12, 13] which relied on the Larch [20] theorem prover. Using Coq allowed us to apply those techniques beyond the query optimizer and prove correct a large subset of the compilation pipeline, including translations between intermediate languages and type checking.

Code Fragments. The third challenge is handling the need for code fragments as preconditions for rewrites. Here, we simply take advantage of the expressive specification language that Coq provides and which can be used to specify and reason about complex conditions (type conditions, ordering conditions, etc.) on the algebraic plans. Take for example the distinct elimination law:

```
Lemma tdup_elim q : nodupA q -> ♯distinct(q) ⇒ q.          ✿
Proof. ... Qed.
```

The predicate `nodupA q` holds when the query plan `q` always returns a collection without duplicates, and `->` is the syntax for logical implication in Coq. In contrast to traditional rule-based optimizers, the `nodupA` predicate is not a subroutine written in a traditional programming language but is written in Coq itself and has also been proved correct.

Overview. The next section illustrates the distinction between variable-based and combinator-based algebras, how that distinction impacts algebraic equivalences, and introduces NRA^e through examples. The rest of the paper contains the formal treatment for the proposed NRA extension and its properties, applications, and implementation. This paper makes the following main contributions:

- It describes a new approach to handling environments in database algebras and defines NRA^e, an extension of a combinator-based nested relational algebra with support for environments (Section 3).
- It extends the traditional notion of algebraic equivalence for NRA^e and defines algebraic rewrites for environment manipulation. A main result is that all existing algebraic equivalences for the original combinator-based NRA can be lifted "as is" to NRA^e (Section 4).
- It shows that NRA^e can be effectively compiled back to traditional NRA and calculus. This confirms that NRA^e has the expected expressiveness and provides a bridge for integration into existing systems (Section 5).
- It illustrates NRA^e on several use cases. We show it can naturally encode an equivalent NRA with lambda terms. We also show how environment operators provide an elegant way to represent view declarations in SQL or OQL (Section 6). Finally, we used NRA^e to radically simplify optimization for a query DSL built on top of JRules [8] (Section 7).

There is more to a query compiler than its core algebra and optimizer. In Section 8, we report on the status of our effort in building Q*cert, an end-to-end, formally verified, query compiler based on NRA^e. We review aspects that were left out of the main formal treatment in the paper, notably: front-end support, code generation, type checking and handling of user-defined types and functions.

Although not necessary to follow the paper, the reader can consult the full compiler specification which we have made available at https://querycert.github.io/sigmod17.

2. VARIABLES REVISITED

The treatment of variables and scoping in compilers is notoriously challenging, and a wide range of techniques have been proposed to encode variables in a way that facilitates reasoning, either for correctness or optimization purposes [5]. The topic has received less attention in the database context. One area where issues related to variable handling come to the fore is rule-based optimizers [10, 12, 13, 18, 24,

$$\begin{array}{ll}
\textbf{T1 (lambdas):} & \mathbf{map}(\lambda a.(a.city))(\mathbf{map}(\lambda p.(p.addr))(P)) \equiv \mathbf{map}(\lambda p.((p.addr).city))(P) \\
\textbf{T1}^c \textbf{ (combin.):} & \chi_{\langle \mathbf{In}.a.city \rangle}\left(\chi_{\langle [a:\mathbf{In}] \rangle}\left(\chi_{\langle \mathbf{In}.p.addr \rangle}\left(\chi_{\langle [p:\mathbf{In}] \rangle}(P)\right)\right)\right) \equiv \chi_{\langle \mathbf{In}.p.addr.city \rangle}\left(\chi_{\langle [p:\mathbf{In}] \rangle}(P)\right) \\
\textbf{T1}^e \textbf{ (NRA}^e\textbf{):} & \chi_{\langle \mathbf{Env}.a.city \circ^e [a:\mathbf{In}] \rangle}\left(\chi_{\langle \mathbf{Env}.p.addr \circ^e [p:\mathbf{In}] \rangle}(P)\right) \equiv \chi_{\langle \mathbf{Env}.p.addr.city \circ^e [p:\mathbf{In}] \rangle}(P)
\end{array}$$

$$\begin{array}{ll}
\textbf{A4 (lambdas):} & \mathbf{map}(\lambda p.([person : p,\ kids : \mathbf{filter}(\lambda c.(p.age > 25))(p.child)])(P) \\
\textbf{A4}^c \textbf{ (combin.):} & \chi_{\left\langle \left[person : \mathbf{In}.p,\ kids : \sigma_{\langle \mathbf{In}.p.age > 25 \rangle}\left(\bowtie^d_{\langle \chi_{\langle [c:\mathbf{In}] \rangle}(\mathbf{In}.p.child) \rangle}(\{\mathbf{In}\}) \right) \right] \right\rangle}\left(\chi_{\langle [p:\mathbf{In}] \rangle}(P)\right) \\
\textbf{A4}^e \textbf{ (NRA}^e\textbf{):} & \chi_{\langle [person : \mathbf{Env}.p,\ kids : \sigma_{\langle (\mathbf{Env}.p.age > 25) \circ^e (\mathbf{Env} \oplus [c:\mathbf{In}]) \rangle}(\mathbf{Env}.p.child)] \ \circ^e \ [p : \mathbf{In}] \rangle}(P)
\end{array}$$

Figure 1: Three styles of nested relational algebra for **T1** and **A4** (Examples from Cherniack and Zdonik [12])

31]. Cherniack and Zdonik clearly describe the challenges caused by variables in database algebras and propose a solution based on combinators [12]. Figure 1 shows two examples from that paper. We use $e.a$ (resp. $[a_1 : e_1, \ldots, a_n : e_n]$) to denote record access (resp. record construction).

Lambdas vs. Combinators. Most internal database algebras support some form of explicit binders, most commonly expressed as lambdas [10, 12, 18]. Example **T1** in Figure 1 shows an equivalence written in AQUA [24] that illustrates how lambdas are used inside iterators with **map** (resp. **filter**) corresponding to functional map (resp. selection).

Both query plans in **T1** return the content of the city fields within the addr fields in the records returned by P. The rewrite is a classic loop fusion: in lambda form, it can be expressed using beta-reduction (or capture avoiding substitution) as suggested by Fegaras et al. [18]:

$$\mathbf{map}(\lambda x.(e))(\mathbf{map}(\lambda y.(u))(v)) \equiv \mathbf{map}(\lambda y.(e[u/x]))(v)$$

Although techniques exist (e.g., [1]) to implement or reason about binders and support such substitutions effectively, most of them remain challenging to mechanize and prove correct [5]. As Cherniack and Zdonik first pointed out [12], it is also unnecessarily complex in the database context as equivalent combinator-based algebras can avoid binders and variables. We use Cluet and Moerkotte's algebra [14] as our starting point, because (i) it has been used successfully for the compilation and optimization of nested query languages (notably OQL [14] and XQuery [27]), and (ii) it has already been provided with a complete formalization [34] as combinators. In addition to χ (map) and σ (selection), two important combinators are **In** for accessing the input and $\circ$ for query composition. The following is a combinator-based query plan equivalent to the example **T1**:

$$\textbf{T1'}\text{:}\ \chi_{\langle \mathbf{In}.city \rangle}\left(\chi_{\langle \mathbf{In}.addr \rangle}(P)\right) \equiv \chi_{\langle \mathbf{In}.addr.city \rangle}(P)$$

One immediately notices that lambdas are gone and variables have been replaced by **In** (the implicit input). As Cherniack and Zdonik pointed out, combinators enable algebraic rewrites to be expressed without explicit variable substitution or renaming, and without having to compute (and prove correct) preconditions on the presence/absence of free variables [12]. For instance, the previous map-fusion rewrite can be expressed simply with query composition:

$$\chi_{\langle P_1 \rangle}\left(\chi_{\langle P_2 \rangle}(P)\right) \equiv \chi_{\langle P_1 \circ P_2 \rangle}(P)$$

Reifying Environments. Despite those clear benefits, combinators come at a price: when the query plan actually requires environments containing more than one value, those have to be encoded in the structure of the query plan itself. To illustrate this, consider example **A4** from Figure 1 which features a selection inside a map. Two variables (p and c) are both in scope within the selection predicate. In algebras with combinators, the absence of variables forces environments to be *reified* as records whose fields correspond to the variables in scope.

Figure 1 shows a systematic encoding with reified environments. Compared to **T1'**, $\mathbf{T1}^c$ has an additional map to create a record with field p corresponding to variable p, and accessing that variable has been replaced by $\mathbf{In}.p$ and similarly for variable a. Although this looks relatively innocuous, the additional encoding required for example **A4** is more complex. The initial variable p is reified similarly as in $\mathbf{T1}^c$ and passed as input to the nested plan within the top-level map operator. But adding variable c to that initial environment corresponds to a dependent join (written $\bowtie^d$) combined with a map. The dependent join is an operator introduced by NRA for nested queries, and the semantics resemble a Cartesian product from relational algebra, except that the second operand ($\chi_{\langle [c:\mathbf{In}] \rangle}(\mathbf{In}.p.child)$ in our example) can depend on the value of records returned by the first operand ($\{\mathbf{In}\}$ in our example). Here it is used to build records with both p and c fields, encoding the addition of variable c to the environment. Despite the many techniques developed for optimization of nested-relational plans, the use of a dependent join and the additional nesting is a heavy price to pay for such a simple example. In our experience such encoding can inhibit optimization, making the query optimizer harder to develop and to apply in practical scenarios.

NRA^e. To address those shortcomings, we define NRA^e, an extension of a nested relational algebra with combinators that includes specific operators for environment manipulation. It keeps the benefits of the combinator-based approach for reasoning, but simplifies the encoding of source queries, making existing optimization techniques more effective in practice. The main intuition for the extension is as follows: instead of using combinators with one implicit input (the current value **In**), NRA^e uses combinators with two implicit inputs (one for the current value **In** and one for the reified environment **Env**). To illustrate that idea, let us look again at example **T1** and the equivalent formulation $\mathbf{T1}^e$ written with NRA^e. The environment is reified similarly as $\mathbf{T1}^c$ with a record containing a field p, but that environment is passed using a special combinator $\circ^e$, which sets the environment part of the input. Once the environment has been set, it can be accessed using the **Env** combinator. Similarly, the encoding for **A4** avoids additional nested maps and join operations. It sets the environment with $\circ^e$ and extends the environment with record concatenation ($\mathbf{Env} \oplus [c : \mathbf{In}]$).

In the rest of the paper, we formally define NRA^e, study its formal properties, and illustrate its use in practice.

3. NRA^e

This section introduces NRA^e, our extension of NRA that supports environments. To do so, we first define a data model for complex values, operators on that data model, and a combinators-based NRA.

3.1 Data Model and Operators

Values in our data model are constants, bags, or records ✿:

$$d ::= c \mid \emptyset \mid \{\overline{d_i}\} \mid [\,] \mid [\overline{A_i : d_i}]$$

Constants (c) includes integers, strings, etc. A bag is a multiset of values. Let $\emptyset$ denote the empty bag and $\{d_1, ..., d_n\}$ the bag with values $d_1, ..., d_n$. A record is a mapping from a finite set of attributes to values, where attribute names are drawn from a sufficiently large set $A, B, ...$ Let $[\,]$ denote the empty record and $[\overline{A_i : d_i}]$ the record mapping A_i to d_i. $\mathrm{dom}([\overline{A_i : d_i}])$ is the set of labels A_i.

Unary and binary operators are basic operations over the data model. Unary operators include the following ✿✿:

$\boxplus\, d ::= \mathit{ident}\ d$	returns d
$\mid\ \neg d$	negates a Boolean
$\mid\ \{d\}$	the singleton bag containing d
$\mid\ \mathit{flatten}\ d$	flattens one level of a bag of bags
$\mid\ [A : d]$	the record with attribute A of value d
$\mid\ d.A$	the value of attribute A in record d
$\mid\ d - A$	removes attribute A from record d
$\mid\ \pi_{\overline{A_i}}(d)$	the projection of record d over $\overline{A_i}$

Binary operators include the following ✿✿:

$d_1 \boxtimes d_2 ::= d_1 = d_2$	compares two data for equality
$\mid\ d_1 \in d_2$	true if d_1 is an element of bag d_2
$\mid\ d_1 \cup d_2$	the union of two bags
$\mid\ d_1 \oplus d_2$	concatenates two records, favoring d_2 for overlapping attributes
$\mid\ d_1 \otimes d_2$	returns a singleton with the record concatenation if compatible, and $\emptyset$ otherwise

Compatibility-based concatenation is used to capture unification in binders (with the same semantics as in a natural join). Two records x and y are deemed *compatible* if common attributes match: $\forall A \in \mathrm{dom}(x) \cap \mathrm{dom}(y)$, $x(A) = y(A)$.

We only gave here a few key operators, but those can be easily extended (e.g, for arithmetic or aggregation). The record operations are sufficient to support all the classic relational and nested relational operators.

3.2 Combinator-based NRA

We first give a definition for the combinator-based NRA which is the basis for our extension.

Definition 1 (NRA syntax ✿).

$$\begin{aligned} q \quad ::= \quad & d \mid \mathbf{In} \mid q_2 \circ q_1 \mid \boxplus q \mid q_1 \boxtimes q_2 \mid \chi_{\langle q_2 \rangle}(q_1) \\ & \mid \sigma_{\langle q_2 \rangle}(q_1) \mid q_1 \times q_2 \mid \bowtie^d{}_{\langle q_2 \rangle}(q_1) \mid q_1 \parallel q_2 \end{aligned}$$

This algebra is the one from [14, 34]. Most operators should be familiar to the reader, with the exception of $\parallel$, which was introduced in [34] to handle aspects of error propagation and which will be needed in Section 7.

Here, d returns constant data, $\mathbf{In}$ returns the context value (usually a bag or a record), and $q_2 \circ q_1$ denotes query plan composition, i.e., it evaluates q_2 using the result of q_1 as input value. $\boxplus$ and $\boxtimes$ are unary and binary operators from Section 3.1. χ is the map operation on bags, σ is selection, and $\times$ is the Cartesian product. The *dependent join* $\bowtie^d{}_{\langle q_2 \rangle}(q_1)$ evaluates q_2 with its context set to each value in the bag resulting from evaluating q_1, then concatenates records from q_1 and q_2 as in a Cartesian product. The $\parallel$ expression, called *default*, evaluates its first operand and returns its value, unless that value is $\emptyset$, in which case it returns the value of its second operand (as default).

Note that other NRA operators useful for optimization (e.g., joins or group-by) can be defined in terms of this core algebra. For example, the standard relational projection is defined as $\Pi_{\overline{A_i}}(q) = \chi_{\left\langle \pi_{\overline{A_i}} \right\rangle}(q)$ ✿, and unnest, which will be used in Section 5, is defined as:

$$\rho_{B/\{A\}}(q) = \chi_{\langle \mathbf{In} - A \rangle}\left(\bowtie^d{}_{\langle \chi_{\langle [B:\mathbf{In}] \rangle}(\mathbf{In}.A) \rangle}(q) \right) ✿$$

3.3 NRA^e Syntax and Semantics

The following gives the syntax for NRA^e, which is a proper extension from the combinator-based NRA from Section 3.2.

Definition 2 (NRA^e syntax ✿).

$$q \quad ::= \quad ... \mid \mathbf{Env} \mid q_2 \circ^e q_1 \mid \chi^e_{\langle q \rangle}$$

We denote by NRA(q) the property that query q does not use any of the new operators ✿: the set of plans q such that NRA(q) is the standard NRA. Let $\mathcal{I}^e(q)$ ✿ (resp. $\mathcal{I}^i(q)$ ✿) denote the property that query plan q ignores the environment $\mathbf{Env}$ (resp. the input data $\mathbf{In}$).

Figure 2 gives an operational semantics for NRA^e. It is defined by a judgment of the form $\gamma \vdash q \mathbin{@} d \Downarrow_a d'$ which reads as: in the environment γ, the query q is evaluated against the input data d and produces output data d'. The environment γ can be any value, but in most cases, it is a record whose fields correspond to variable bindings. The rules for the NRA constructs of NRA^e are the same as the rules for $\vdash q \mathbin{@} d \Downarrow_a d'$ used to define the semantics of NRA in [34].

Three operators are added to manipulate environments: access to the environment ($\mathbf{Env}$), composition over the environment ($q_2 \circ^e q_1$), and map over the environment ($\chi^e_{\langle q \rangle}$). The semantics of $\mathbf{Env}$ is to return the current environment. Hence, for example, if we want to access the value of the variable A in the environment, we can write $\mathbf{Env}.A$.

The semantics of $q_2 \circ^e q_1$ is to evaluate q_2 in the environment bound to the value returned by q_1. It is similar to query composition ($\circ$) but changes the environment rather than the input value. This construct is useful for example to add a value d in the environment associated to the variable A for the evaluation of a query q: $q \ \circ^e \ (\mathbf{Env} \oplus [A : d])$, keeping in mind that the record concatenation operator $\oplus$ favors the right-most binding in case of conflict.

The last operator, χ^e, is dual to the standard map but iterates over the environment rather than over the input collection. It is mainly useful to handle the result of merging two environments using the $\otimes$ operator. The expression: $\chi^e_{\langle q \rangle} \ \circ^e \ (\mathbf{Env} \otimes [A : d])$ merges a new binding for variable A with value d to an existing environment, and passes the

$$\frac{}{\gamma \vdash d_0 @ d \Downarrow_a d_0}\ \text{(Constant)} \qquad \frac{}{\gamma \vdash \mathbf{In} @ d \Downarrow_a d}\ \text{(ID)} \qquad \frac{\gamma \vdash q_1 @ d_0 \Downarrow_a d_1 \quad \gamma \vdash q_2 @ d_1 \Downarrow_a d_2}{\gamma \vdash q_2 \circ q_1 @ d_0 \Downarrow_a d_2}\ \text{(Comp)}$$

$$\frac{\gamma \vdash q @ d \Downarrow_a d_0 \quad \boxplus d_0 = d_1}{\gamma \vdash \boxplus q @ d \Downarrow_a d_1}\ \text{(Unary)} \qquad \frac{\gamma \vdash q_1 @ d \Downarrow_a d_1 \quad \gamma \vdash q_2 @ d \Downarrow_a d_2 \quad \gamma \vdash d_1 \boxtimes d_2 = d_3}{\gamma \vdash q_1 \boxtimes q_2 @ d \Downarrow_a d_3}\ \text{(Binary)}$$

$$\frac{\gamma \vdash q_1 @ d \Downarrow_a \emptyset}{\gamma \vdash \chi_{\langle q_2 \rangle}(q_1) @ d \Downarrow_a \emptyset}\ \text{(Map } \emptyset\text{)} \qquad \frac{\gamma \vdash q_1 @ d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash q_2 @ d_1 \Downarrow_a d_2 \quad \gamma \vdash \chi_{\langle q_2 \rangle}(s_1) @ d \Downarrow_a s_2}{\gamma \vdash \chi_{\langle q_2 \rangle}(q_1) @ d \Downarrow_a \{d_2\} \cup s_2}\ \text{(Map)}$$

$$\frac{\gamma \vdash q_1 @ d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash q_2 @ d_1 \Downarrow_a \mathsf{true} \quad \gamma \vdash \sigma_{\langle q_2 \rangle}(s_1) @ d \Downarrow_a s_2}{\gamma \vdash \sigma_{\langle q_2 \rangle}(q_1) @ d \Downarrow_a \{d_1\} \cup s_2}\ (\text{Sel}_{\text{T}})$$

$$\frac{\gamma \vdash q_1 @ d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash q_2 @ d_1 \Downarrow_a \mathsf{false} \quad \gamma \vdash \sigma_{\langle q_2 \rangle}(s_1) @ d \Downarrow_a s_2}{\gamma \vdash \sigma_{\langle q_2 \rangle}(q_1) @ d \Downarrow_a s_2}\ (\text{Sel}_{\text{F}})$$

$$\frac{\gamma \vdash q_1 @ d \Downarrow_a \emptyset}{\gamma \vdash \sigma_{\langle q_2 \rangle}(q_1) @ d \Downarrow_a \emptyset}\ (\text{Sel}_{\emptyset}) \qquad \frac{\gamma \vdash q_1 @ d \Downarrow_a \emptyset}{\gamma \vdash q_1 \times q_2 @ d \Downarrow_a \emptyset}\ (\text{Prod}^l_{\emptyset}) \qquad \frac{\gamma \vdash q_2 @ d \Downarrow_a \emptyset}{\gamma \vdash q_1 \times q_2 @ d \Downarrow_a \emptyset}\ (\text{Prod}^r_{\emptyset})$$

$$\frac{\gamma \vdash q_1 @ d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash q_2 @ d \Downarrow_a \{d_2\} \cup s_2 \quad \gamma \vdash \{d_1\} \times s_2 @ d \Downarrow_a s_3 \quad \gamma \vdash s_1 \times (\{d_2\} \cup s_2) @ d \Downarrow_a s_4}{\gamma \vdash q_1 \times q_2 @ d \Downarrow_a \{d_1 \oplus d_2\} \cup s_3 \cup s_4}\ \text{(Prod)}$$

$$\frac{\gamma \vdash q_1 @ d \Downarrow_a \{d_1\} \cup s_1 \quad \gamma \vdash \{d_1\} \times q_2 @ d_1 \Downarrow_a s_2 \quad \gamma \vdash {\bowtie^d}_{\langle q_2 \rangle}(s_1) @ d \Downarrow_a s_3}{\gamma \vdash {\bowtie^d}_{\langle q_2 \rangle}(q_1) @ d \Downarrow_a s_2 \cup s_3}\ \text{(DJ)} \qquad \frac{\gamma \vdash q_1 @ d \Downarrow_a \emptyset}{\gamma \vdash {\bowtie^d}_{\langle q_2 \rangle}(q_1) @ d \Downarrow_a \emptyset}\ (\text{DJ}_{\emptyset})$$

$$\frac{\gamma \vdash q_1 @ d \Downarrow_a d_1 \quad d_1 \neq \emptyset}{\gamma \vdash q_1 \parallel q_2 @ d \Downarrow_a d_1}\ (\text{Default}_{\neg\emptyset}) \qquad \frac{\gamma \vdash q_1 @ d \Downarrow_a \emptyset \quad \gamma \vdash q_2 @ d \Downarrow_a d_2}{\gamma \vdash q_1 \parallel q_2 @ d \Downarrow_a d_2}\ (\text{Default}_{\emptyset})$$

$$\frac{}{\gamma \vdash \mathbf{Env} @ d \Downarrow_a \gamma}\ \text{(Env)} \qquad \frac{\gamma_1 \vdash q_1 @ d_1 \Downarrow_a \gamma_2 \quad \gamma_2 \vdash q_2 @ d_1 \Downarrow_a d_2}{\gamma_1 \vdash q_2 \circ^e q_1 @ d_1 \Downarrow_a d_2}\ (\text{Comp}^e)$$

$$\frac{}{\emptyset \vdash \chi^e_{\langle q_2 \rangle} @ d \Downarrow_a \emptyset}\ (\text{Map}^e_{\emptyset}) \qquad \frac{d_1 \vdash q_2 @ d \Downarrow_a d_2 \quad s_1 \vdash \chi^e_{\langle q_2 \rangle} @ d \Downarrow_a s_2}{\{d_1\} \cup s_1 \vdash \chi^e_{\langle q_2 \rangle} @ d \Downarrow_a \{d_2\} \cup s_2}\ (\text{Map}^e)$$

Figure 2: NRA^e Semantics ✿. $\boxed{\gamma \vdash q @ d \Downarrow_a d}$

resulting environment (if successful) to the subsequent query q. Let us assume the environment **Env** contains the record $[A:1, B:3]$, the following shows an example where merge succeeds (resp. fails) over the common variable B:

$$\chi^e_{\langle \mathbf{Env}.A + \mathbf{Env}.C \rangle} \circ^e (\mathbf{Env} \otimes [B:3, C:4]) \Rightarrow \{5\} \quad ✿$$
$$\chi^e_{\langle \mathbf{Env}.A + \mathbf{Env}.C \rangle} \circ^e (\mathbf{Env} \otimes [B:2, C:4]) \Rightarrow \{\} \quad ✿$$

After the merge, the environment contains a collection, in our example either $\{[A:1, B:3, C:4]\}$ or $\{\}$, and χ^e accounts for that. This feature is an important advantage of the combinators-based approach over a lambda-based approach: it allows it to capture environment unification which is notably useful for rule-based languages, e.g., in Sparql or the CAMP calculus in Section 7.

4. OPTIMIZATION

This section presents rewrites designed to optimize NRA^e query plans. We also prove that any existing NRA rewrites can be applied to an NRA^e query plan, even if subexpressions manipulate the environment. First, we define a notion of equivalence to capture rewrite correctness.

4.1 Equivalences

The semantics of equivalences we use to define and prove correctness follows the classic notion of *strong equivalence* as defined in [2] (rather than *weak equivalence*).

Definition 3 (Equivalence ✿). *Two plans q_1 and q_2 are equivalent ($q_1 \equiv q_2$) iff for any environment γ and for any input data d, evaluating q_1 and q_2 over data d in environment γ returns the same value. I.e.,* $\forall \gamma, d$

$$(\exists d_1, \gamma \vdash q_1 @ d \Downarrow_a d_1 \iff \exists d_2, \gamma \vdash q_2 @ d \Downarrow_a d_2) \wedge$$
$$\forall d_1, d_2, (\gamma \vdash q_1 @ d \Downarrow_a d_1) \wedge (\gamma \vdash q_2 @ d \Downarrow_a d_2) \implies d_1 = d_2$$

As in most database optimizers, we only consider rewriting for well-typed algebraic plans. In our context, we focus on directed equivalences, where the direction indicates the way those are used in the optimizer. Since we have omitted treatment of type checking from the paper, we leave that definition somewhat informal.

Definition 4 (Typed Rewrites ✿). *We say a query plan q_1 rewrites to query plan q_2, written $q_1 \Rightarrow q_2$ iff, given a well-typed q_1, then q_2 is also well typed, and for all well-typed input data and environments, they return the same value.*

The corresponding equivalence and typed rewrite relation are defined similarly for NRA ✿✿.

Both plan equivalence and typed rewrites are contextual: given any plan C_1, and a sub-plan q which is a sub-expression of C, replacing q_1 by q_2 (where $q_1 \equiv q_2$ or $q_1 \Rightarrow q_2$) yields a new C_2 such that $C_1 \equiv C_2$ or $C_1 \Rightarrow C_2$ as appropriate. In the mechanization, this is expressed as a set of proofs that each type of expression preserves plan equivalence and typed rewrites. For example, swapping two equivalent sub-plans

Environment constructs removal

$$q \circ^e \mathbf{Env} \Rightarrow q \quad ✿$$
$$\mathbf{Env} \circ^e q \Rightarrow q \quad ✿$$
$$\text{if } \mathcal{I}^e(q_1),\ q_1 \circ^e q_2 \Rightarrow q_1 \quad ✿$$
$$\chi_{\langle \mathbf{Env} \rangle}\big(\sigma_{\langle q \rangle}(\{\mathbf{In}\})\big) \circ^e \mathbf{In} \Rightarrow \sigma_{\langle q \rangle}(\{\mathbf{In}\}) \circ^e \mathbf{In} \quad ✿$$
$$\text{if } \mathcal{I}^e(q_1),\ \chi_{\langle \mathbf{Env} \rangle}\big(\sigma_{\langle q_1 \rangle}(\{\mathbf{In}\})\big) \circ^e q_2 \Rightarrow \chi_{\langle q_2 \rangle}\big(\sigma_{\langle q_1 \rangle}(\{\mathbf{In}\})\big) \quad ✿$$
$$\chi^e_{\langle \mathbf{Env} \rangle} \circ q \Rightarrow \mathbf{Env} \quad ✿$$
$$\chi^e_{\langle q_1 \rangle} \circ^e \{q_2\} \Rightarrow \{q_1 \circ^e q_2\} \quad ✿$$
$$\text{if } \mathcal{I}^i(q_1),\ \chi^e_{\langle q_1 \rangle} \circ^e q_2 \Rightarrow \chi_{\langle q_1 \circ^e \mathbf{In} \rangle}(q_2) \quad ✿$$

$\circ^e$ pushdown

$$(\boxplus(q_1)) \circ^e q_2 \Rightarrow \boxplus(q_1 \circ^e q_2) \quad ✿$$
$$(q_1 \boxtimes q_2) \circ^e q \Rightarrow (q_1 \circ^e q) \boxtimes (q_2 \circ^e q) \quad ✿$$
$$\text{if } \mathcal{I}^i(q),\ \chi_{\langle q_1 \rangle}(q_2) \circ^e q \Rightarrow \chi_{\langle q_1 \circ^e q \rangle}(q_2 \circ^e q) \quad ✿$$
$$\text{if } \mathcal{I}^i(q),\ \sigma_{\langle q_1 \rangle}(q_2) \circ^e q \Rightarrow \sigma_{\langle q_1 \circ^e q \rangle}(q_2 \circ^e q) \quad ✿$$
$$(q_1 \circ^e q_2) \circ^e q \Rightarrow q_1 \circ^e (q_2 \circ^e q) \quad ✿$$
$$\text{if } \mathcal{I}^e(q_1),\ (q_1 \circ q_2) \circ^e q \Rightarrow q_1 \circ (q_2 \circ^e q) \quad ✿$$
$$\text{if } \mathcal{I}^e(q_1),\ q_1 \circ^e q_2 \Rightarrow q_1 \quad ✿$$
$$\text{if } \mathcal{I}^e(q_1),\ (\mathbf{Env} \otimes q_1) \circ^e q \Rightarrow q \otimes q_1 \quad ✿$$
$$\sigma_{\langle q \rangle}(\{\mathbf{In}\}) \circ^e \mathbf{In} \Rightarrow \sigma_{\langle q \circ^e \mathbf{In} \rangle}(\{\mathbf{In}\}) \quad ✿$$

Figure 3: Rewrites for NRA^e

inside a map operator yields an equivalent plan ✿✿. These proofs, when taken together, establish that plan equivalence and typed rewrites are contextual, and enables rewriting sub-expressions freely, which is critical for optimization.

Note that, as in the relational context, plan equivalence implies typed rewrites as long as the result is well-typed (assuming the source is) ✿✿. Our implementation includes a full type checker and the correctness proofs for all the rewrites used in the optimizer have been verified for both untyped and typed cases (depending on the specific rewrites).

4.2 Lifting NRA Rewrites

One of the most important properties of NRA^e is the ability to reuse existing known equivalences from NRA. This is a strong result since we allow lifting equivalences over query plans that may contain environment manipulations. To illustrate that idea, let us consider a simple selection pushdown equivalence from the relational literature:

$$\sigma_{\langle q_0 \rangle}(q_1 \cup q_2) \equiv \sigma_{\langle q_0 \rangle}(q_1) \cup \sigma_{\langle q_0 \rangle}(q_2)$$

For NRA, q_0 is an arbitrary query plan returning a Boolean value. Our lifting result shows that if such an equivalence is true for any well typed q_0, q_1, q_2 in NRA, then it is also true for any well typed q_0, q_1, q_2 in NRA^e.

This result relies on the fact that the NRA equivalences are effectively a form of parametric polymorphism in terms of q_0, q_1, q_2. To properly express this, we need a stronger notion of equivalence which is parametric. We first define parametric plans for both the NRA and NRA^e, which are query plans with some *plan variables*, along with parametric plan *instantiation*. We use the set $\mathcal{PV} = \{\$q_0, \ldots, \$q_n\}$ to denote plan variables.

Definition 5 (Parametric Plan ✿✿). *A parametric plan c over plan variables $\$q_0, \ldots, \$q_n$ is an expression in the NRA (resp. NRA^e) grammar extended with plan variables $\$q_0, \ldots, \$q_n$.*

For example, $\sigma_{\langle \$q_0 \rangle}(\$q_1 \cup \$q_2)$ denotes a parametric plan over plan variables $\$q_0$, $\$q_1$, $\$q_2$.

Definition 6 (Plan Instantiation ✿✿). *Given a parametric plan c over plan variables $\$q_0, \ldots, \$q_n$, the instantiation of c with $q_0, \ldots, q_n$, denoted $c[q_0, \ldots, q_n]$, is the query plan obtained by substituting $\$q_i$ by q_i in c.*

We can now define parametric equivalence, which states that two parametric plans are equivalent if every plan instantiation for those two plans are equivalent.

Definition 7 (Parametric equiv. ✿✿). *Given two parametric plans c_1 and c_2 over $\$q_1, \ldots, \$q_n$, we say that they are parametric equivalent iff, for every plans $q_1, \ldots, q_n$:*

$$c_1[q_0, \ldots, q_n] \equiv c_2[q_0, \ldots, q_n]$$

We use $c_1 \equiv_c c_2$ and $c_1 \equiv^e_c c_2$ to denote parametric equivalence for the NRA and NRA^e respectively.

For example, the following holds for the NRA:

$$\sigma_{\langle \$q_0 \rangle}(\$q_1 \cup \$q_2) \equiv_c \sigma_{\langle \$q_0 \rangle}(\$q_1) \cup \sigma_{\langle \$q_0 \rangle}(\$q_2) \quad ✿$$

Most relational or nested relational equivalences are in fact parametric. Formalizing parametric equivalence enables the precise statement of the following key lifting theorem:

Theorem 1 (Equiv. Lifting ✿). *Every parametric NRA equivalence is also a parametric NRA^e equivalence:*

$$c_1 \equiv_c c_2 \implies c_1 \equiv^e_c c_2$$

This result and corresponding proof are non-trivial and deserve a few comments. First, recall that every NRA operator is also an NRA^e operator. This means the theorem statement is well-formed in the sense that the operators in c_1 and c_2 are also NRA^e operators that can be used on the right-hand side. Second, the proof fundamentally relies on the ability to translate NRA^e back to NRA (Theorem 2 in Section 5). It also relies on the observation that the part of the query that was lifted from NRA cannot *change* the environment. The instantiated NRA^e expressions can interact with the environment, but modifications are local. The proof can thus treat the environment as mostly constant.

4.3 NRA^e Rewrites

In addition to NRA optimizations lifted to NRA^e, we developed additional optimizations for our extended algebra. We report on two categories of rewrites, which are given in Figure 3. The first category contains rewrites that remove environment manipulation constructs. For example, in the first rewrite ($q \circ^e \mathbf{Env} \Rightarrow q$), it is possible to get rid of the composition over the environment because it replaces the value of the environment for the evaluation of q by itself.

Some of the rewrites use the predicates introduced in Section 3.3 that test if a query ignores the context data ($\mathcal{I}^i(q)$) or the environment ($\mathcal{I}^e(q)$). For example, the third rewrite (if $\mathcal{I}^e(q_1)$, $q_1 \circ^e q_2 \Rightarrow q_1$) shows that if the query on the left of a composition over the environment does not access the value of the environment, then it is not necessary to replace the value of the environment by the value of q_2. Like the `nodupA` example in the introduction, the $\mathcal{I}^i(q)$ and $\mathcal{I}^e(q)$

$$
\begin{aligned}
[\![d]\!]_a &= d\\
[\![\mathbf{In}]\!]_a &= \mathbf{In}.D\\
[\![q_2 \circ q_1]\!]_a &= [\![q_2]\!]_a \circ ([E : \mathbf{In}.E] \oplus [D : [\![q_1]\!]_a])\\
[\![\boxplus q]\!]_a &= \boxplus [\![q]\!]_a\\
[\![q_1 \boxtimes q_2]\!]_a &= [\![q_1]\!]_a \boxtimes [\![q_2]\!]_a\\
[\![\chi_{\langle q_2\rangle}(q_1)]\!]_a &= \chi_{\langle [\![q_2]\!]_a\rangle}\big(\rho_{D/\{T_1\}}\left(\{[E : \mathbf{In}.E] \oplus [T_1 : [\![q_1]\!]_a]\}\right)\big)\\
[\![\sigma_{\langle q_2\rangle}(q_1)]\!]_a &= \chi_{\langle \mathbf{In}.D\rangle}\big(\sigma_{\langle [\![q_2]\!]_a\rangle}\big(\rho_{D/\{T_1\}}\left(\{[E : \mathbf{In}.E] \oplus [T_1 : [\![q_1]\!]_a]\}\right)\big)\big)\\
[\![q_1 \times q_2]\!]_a &= [\![q_1]\!]_a \times [\![q_2]\!]_a\\
[\![\bowtie^d{}_{\langle q_2\rangle}(q_1)]\!]_a &= \chi_{\langle \mathbf{In}.D \oplus \mathbf{In}.T_2\rangle}\Big(\bowtie^d{}_{\langle \chi_{\langle [T_2:\mathbf{In}]\rangle}([\![q_2]\!]_a)\rangle}\big(\rho_{D/\{T_1\}}\left(\{[E : \mathbf{In}.E] \oplus [T_1 : [\![q_1]\!]_a]\}\right)\big)\Big)\\
[\![q_1 \| q_2]\!]_a &= [\![q_1]\!]_a \| [\![q_2]\!]_a\\
[\![\mathbf{Env}]\!]_a &= \mathbf{In}.E\\
[\![q_2 \circ^e q_1]\!]_a &= [\![q_2]\!]_a \circ ([E : [\![q_1]\!]_a] \oplus [D : \mathbf{In}.D])\\
[\![\chi^e_{\langle q_2\rangle}]\!]_a &= \chi_{\langle [\![q_2]\!]_a\rangle}\big(\rho_{E/\{T_1\}}\left(\{[T1 : \mathbf{In}.E] \oplus [D : \mathbf{In}]\}\right)\big)
\end{aligned}
$$

Figure 4: From NRA^e to NRA ✿. $\boxed{[\![q]\!]_a = q'}$

predicates are written in Coq and proved correct, showcasing the ability to use code fragments in pre-conditions.

The $\circ^e$ pushdown category is central to the processing of the context. It corresponds to changing the scope for the environment. The general idea here is to push down the context close to the place where it is being used in order to eliminate it, which happens when the composition reaches a leaf. For instance if $\circ^e$ gets pushed down all the way to an **In** it can be eliminated since the environment is not used.

5. TRANSLATING FROM NRA^e

This section defines the translations of NRA^e to NRA and to the Named Nested Relational Calculus (NNRC). The first translation is useful to show that NRA^e shares the same expressiveness as NRA, which is desirable since it establishes that we have not inadvertently targeted a more expressive language. The second translation provides a useful bridge to a representation with variables which can prove useful e.g., for code generation. We will come back to that second point in Section 8 where we describe our implementation.

From NRA^e to NRA. We first consider the relationship between NRA^e and NRA. Figure 4 defines the translation function $[\![q]\!]_a$ from NRA^e to NRA. It relies on the encoding of the two inputs of NRA^e (**In** and **Env**) as a record with two fields: D for the input datum and E for the environment. This record is the single input **In** of NRA. Therefore, the translation of **In** (resp. **Env**) corresponds to accessing field D (resp. E).

This encoding surfaces in the translation of most NRA^e constructs. For example, the translation of composition needs to reconstruct the encoding of the input before the evaluation of the second part of the query:

$$[\![q_2 \circ q_1]\!]_a = [\![q_2]\!]_a \circ ([E : \mathbf{In}.E] \oplus [D : [\![q_1]\!]_a])$$

This translation lifts each element of the collection which is mapped into a record containing the environment as field E and the element of the collection as field D.

The translation uses the unnest operator $\rho_{B/\{A\}}(q)$ defined in Section 3.2. Unsurprisingly, this re-introduces some of the nesting/complexity that was eliminated by supporting environments in NRA^e. The correctness of the translation is established by Theorem 2.

Theorem 2 (NRA^e to NRA Correctness ✿).

$$\gamma \vdash q \mathbin{@} d_1 \Downarrow_a d_2 \iff \vdash [\![q]\!]_a \mathbin{@} ([E : \gamma] \oplus [D : d_1]) \Downarrow_n d_2$$

We mentioned in Section 3.3 that NRA queries have the same behavior evaluated with either NRA or NRA^e semantics ✿. Therefore, in conjunction with Theorem 2, we have a proof that NRA^e has the same expressiveness as NRA.

From NRA^e to NNRC. We present the translation of NRA^e to the Named Nested Relational Calculus (NNRC) [35], with a bag semantics. The syntax of the calculus is ✿:

$$
\begin{array}{rcl}
e & ::= & x \mid d \mid \boxplus e_1 \mid e_1 \boxtimes e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\\
 & \mid & \{e_2 \mid x \in e_1\} \mid e_1\ ?\ e_2 : e_3
\end{array}
$$

Expressions can be variables (x), constants (d), operators ($\boxplus e_1$ or $e_1 \boxtimes e_2$), dependent sequencing ($\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$), bag comprehensions ($\{e_2 \mid x \in e_1\}$), or conditionals ($e_1\ ?\ e_2 : e_3$). The bag comprehension $\{e_2 \mid x \in e_1\}$ constructs a bag where each element is the result of the evaluation of e_2 in an environment in which x is bound to an element of the bag created by the evaluation of e_1. We use the formal semantics of NNRC given in [34] ✿.

Figure 5 defines the translation function $[\![q]\!]_{x_d,x_e}$ from NRA^e to NNRC. The translation function is parameterized by two variables x_d and x_e that are used to encode the input value and the environment. So, for example, the translation of **In** (resp. **Env**) returns the corresponding variable x_d (resp. x_e). This translation makes explicit the handling of the input and the environment. For example, in the translation of the composition the result of the evaluation of the first expression becomes the input of the second expression:

$$[\![q_2 \circ q_1]\!]_{x_d,x_e} = \mathbf{let}\ x = [\![q_1]\!]_{x_d,x_e}\ \mathbf{in}\ [\![q_2]\!]_{x,x_e} \qquad x \text{ is } \mathbf{fresh}$$

The translation of NRA^e to NNRC is similar to the translation of NRA to NNRC presented in [34] ✿. However, the two inputs of NRA^e can be translated directly to NNRC without encoding. Both translations are proved correct ✿✿.

$$
\begin{aligned}
[\![d]\!]_{x_d,x_e} &= d \\
[\![\mathbf{In}]\!]_{x_d,x_e} &= x_d \\
[\![\boxplus q]\!]_{x_d,x_e} &= \boxplus [\![q]\!]_{x_d,x_e} \\
[\![q_1 \boxtimes q_2]\!]_{x_d,x_e} &= [\![q_1]\!]_{x_d,x_e} \boxtimes [\![q_2]\!]_{x_d,x_e} \\
[\![q_2 \circ q_1]\!]_{x_d,x_e} &= \mathbf{let}\ x = [\![q_1]\!]_{x_d,x_e}\ \mathbf{in}\ [\![q_2]\!]_{x,x_e} && x \text{ is } \mathbf{fresh} \\
[\![\chi_{\langle q_2 \rangle}(q_1)]\!]_{x_d,x_e} &= \{[\![q_2]\!]_{x,x_e} \mid x \in [\![q_1]\!]_{x_d,x_e}\} && x \text{ is } \mathbf{fresh} \\
[\![\sigma_{\langle q_2 \rangle}(q_1)]\!]_{x_d,x_e} &= \mathit{flatten}\,(\{[\![q_2]\!]_{x,x_e}\ ?\ \{x\} : \emptyset \mid x \in [\![q_1]\!]_{x_d,x_e}\}) && x \text{ is } \mathbf{fresh} \\
[\![q_1 \times q_2]\!]_{x_d,x_e} &= \mathit{flatten}\,(\{\{x_1 \oplus x_2 \mid x_2 \in [\![q_2]\!]_{x_d,x_e}\} \mid x_1 \in [\![q_1]\!]_{x_d,x_e}\}) && x_1 \text{ is } \mathbf{fresh} \wedge x_2 \text{ is } \mathbf{fresh} \\
[\![\bowtie^d{}_{\langle q_2 \rangle}(q_1)]\!]_{x_d,x_e} &= \mathit{flatten}\,(\{\{x_1 \oplus x_2 \mid x_2 \in [\![q_2]\!]_{x_1,x_e}\} \mid x_1 \in [\![q_1]\!]_{x_d,x_e}\}) && x_1 \text{ is } \mathbf{fresh} \wedge x_2 \text{ is } \mathbf{fresh} \\
[\![q_1 \parallel q_2]\!]_{x_d,x_e} &= \mathbf{let}\ x = [\![q_1]\!]_{x_d,x_e}\ \mathbf{in}\ ((x = \emptyset)\ ?\ [\![q_2]\!]_{x_d,x_e} : x) && x \text{ is } \mathbf{fresh} \\
[\![\mathbf{Env}]\!]_{x_d,x_e} &= x_e \\
[\![q_2 \circ^e q_1]\!]_{x_d,x_e} &= \mathbf{let}\ x = [\![q_1]\!]_{x_d,x_e}\ \mathbf{in}\ [\![q_2]\!]_{x_d,x} && x \text{ is } \mathbf{fresh} \\
[\![\chi^e_{\langle q_2 \rangle}]\!]_{x_d,x_e} &= \{[\![q_2]\!]_{x_d,x} \mid x \in x_e\} && x \text{ is } \mathbf{fresh}
\end{aligned}
$$

Figure 5: From NRA^e to NNRC ✿. $\boxed{[\![q]\!]_{x_d,x_e} = e}$

6. TRANSLATING QUERIES TO NRA^e

We now consider how to use NRA^e as the target for a query language. Because of space considerations, we focus on the following specific aspects: the complexity of the initial translation, the ability to optimize the resulting plan, and the practical benefits of NRA^e's environment support.

NRA$^\lambda$. Our first example is the nested relational algebra with explicit lambdas we used in the introduction to motivate the work. The syntax of NRA$^\lambda$ is the following ✿, where the data model and operators are the same as for NRA^e:

$$
\begin{aligned}
l \ &::= x \mid d \mid \boxplus l \mid l_1 \boxtimes l_2 \mid \mathbf{map}\,(f)\,l \\
&\quad \mid \mathbf{d\text{-}join}\,(f)\,l \mid l_1 \times l_2 \mid \mathbf{filter}\,(f)\,l \\
f \ &::= \lambda x.l
\end{aligned}
$$

The semantics of NRA$^\lambda$ are unsurprising ✿. The main operations behave as in NRA^e, except that dependent operators are expressed explicitly as functions ($\lambda x.l$), which can access their input (as well as any other variables in scope). The scoping rules are standard. Proceeding as for NRA^e, we can define an equivalence relation on NRA$^\lambda$ similar to Definition 3 ✿. This can be used to prove the NRA$^\lambda$ map fusion equivalence given in Figure 1 ✿.

$$
\begin{aligned}
[\![x]\!]_l &= \mathbf{Env}.x \\
[\![d]\!]_l &= d \\
[\![\boxplus l]\!]_l &= \boxplus [\![l]\!]_l \\
[\![l_1 \boxtimes l_2]\!]_l &= [\![l_1]\!]_l \boxtimes [\![l_2]\!]_l \\
[\![\mathbf{map}\,(f)\,l]\!]_l &= \chi_{\langle [\![f]\!]_f \rangle}([\![l]\!]_l) \\
[\![\mathbf{d\text{-}join}\,(f)\,l]\!]_l &= \bowtie^d{}_{\langle [\![f]\!]_f \rangle}([\![l]\!]_l) \\
[\![l_1 \times l_2]\!]_l &= [\![l_1]\!]_l \times [\![l_2]\!]_l \\
[\![\mathbf{filter}\,(f)\,l]\!]_l &= \sigma_{\langle [\![f]\!]_f \rangle}([\![l]\!]_l) \\
[\![\lambda x.l]\!]_f &= [\![l]\!]_l \circ^e (\mathbf{Env} \oplus [x : \mathbf{In}])
\end{aligned}
$$

Figure 6: From NRA$^\lambda$ to NRA^e ✿ $\boxed{[\![l]\!]_l = q \quad [\![f]\!]_f = q}$

The full translation from NRA$^\lambda$ to NRA^e is both small and straightforward and is given in Figure 6. Functions f are translated into an NRA^e expression that adds the current input (the argument to the lambda) to the environment with the appropriate name. The rules for record concatenation correctly enforce local shadowing as needed. Variable lookups are translated into accesses of fields of the environment. This simple translation is easily proved correct (semantics preserving) ✿, validating the suitability of NRA^e for supporting languages with variables. Note that an alternative encoding into NRA would be significantly more complex and harder to prove correct.

This validates the original intuition that we can model traditional variable scoping constructs. It can be useful for adapting rewrites from the literature which often make use of explicit lambda terms [10, 18, 24], or to support language integrated queries written with closures. For instance, the following LINQ [28] expression in C#:

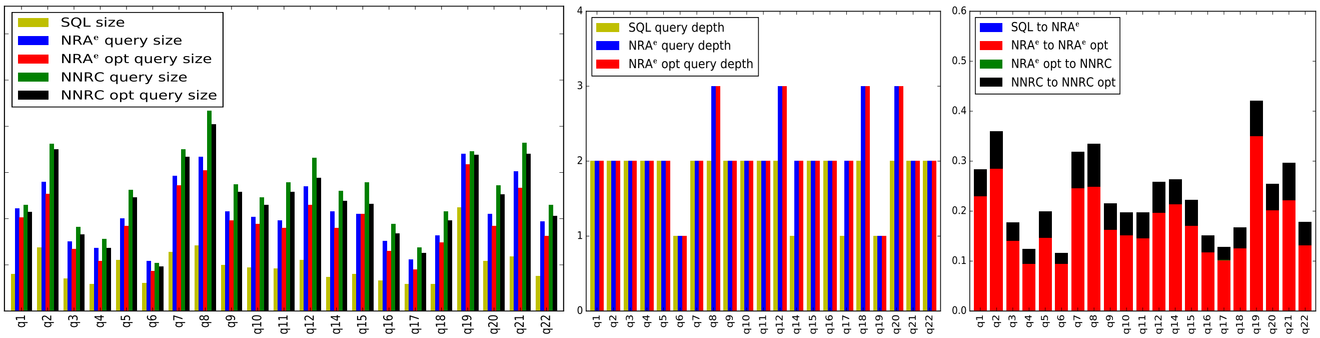
```
Persons.Where(p => p.age < 30).Select(p => p.name)
```

corresponds directly to the NRA$^\lambda$ expression ✿:

$$\mathbf{map}\,(\lambda p.(p.\mathit{name}))\,(\mathbf{filter}\,(\lambda p.(p.\mathit{age} < 30))\,(\mathit{Persons}))$$

SQL. To further evaluate the suitability of NRA^e as a target for query compilation, we implemented a translation from a subset of SQL to NRA^e. A full formal specification for SQL being a large undertaking, our focus here is on validating the compiler and optimizer. That infrastructure relies on an AST for a subset of SQL ✿, along with a translation from that subset to NRA^e ✿. The compiler supports full select-from-where blocks including group by and order by, nested queries, set operations (union, intersect, except), exists, between, view definitions, with clauses, case expressions, comparisons, aggregations, and essential operators on atomic types, including dates and aggregate operations. With that feature set, the compiler handles all TPC-H queries with the exception of one: TPC-H query 13 which uses a left outer join which we currently do not support.[2]

[2]Although our compiler handles null values, we do not have a full specification for the null value semantics in SQL.



(a) SQL, NRA^e, and NNRC query sizes. (b) SQL and NRA^e query depths. Compilation time (s).

Figure 7: TPC-H benchmarks.

We used Q*cert to compile the 21 TPC-H queries that we support, targeting JavaScript as a backend for execution. Since the translation from SQL to NRA^e has not been proven correct, we instead inspected the query results to ensure they were as expected according to the SQL semantics. Figure 7 reports on query size and depth for the NRA^e intermediate representation (with SQL size for comparison), as well as compilation times. The number of operations is relatively large (in the hundreds of operators), but validates that the translation to the algebra does not introduce any unexpected blow up. Compilation time is under two seconds for all queries, with most of the time spent on optimization (translation time is negligible).

As an additional evaluation, we also tried the compiler on TPC-DS queries (without checking for correctness), which are significantly more complex due to some use of rollup and windowing notably. We could compile 37 out of 99 queries, all of which compiled in under 4 seconds except for TPC-DS query 66 which took about 11 seconds. A specific investigation of that query shows a much larger NRA^e plan (around 2200 operators), and most of the compilation time is spent on rewriting.

OQL. The Q*cert compiler also implements a frontend for a reasonable subset of OQL ✿, which is of interest as it provides a clean model for queries over nested relational data and objects, along with aggregation. We implemented the "classic" translation from OQL to NRA proposed in [14] for that subset. That fragment includes select-from-where statements, aggregation, object access, casting and object creation, and arbitrary nesting. We wrote a formal semantics for that fragment ✿ in order to prove the translation to NRA^e correct ✿. Note that most of the translation for OQL does not use environment operators. This is a useful feature of the approach we propose: additional operations on the environment can be used at the discretion of the compiler developer when deemed useful. In the OQL case, all existing NRA optimization [14], e.g., for query decorrelation, can be applied as-is on the resulting plans.

View declarations. One case where we found environment operations to be particularly convenient is in supporting view declaration (and undeclarations). Since we used a similar mechanism for both SQL and OQL views, we illustrate it here only on a SQL example.

The intuition is straightforward: when a view is being defined, it is simply added to the environment with the view name bound to the corresponding query plan. Consider the following simple SQL view definition (inspired by TPC-H query 15):

```
create view revenue0 (supplier_no, total) as
  select l_suppkey, sum(l_extendedprice) from lineitem;
  group by l_suppkey

select s_name, total
from supplier, revenue0
where s_suppkey = supplier_no
  and total = select max(total_revenue) from revenue0;
```

The corresponding translation to NRA^e has the structure $q_{stmt} \circ^e [\texttt{revenue0} : q_{view}]$, where q_{view} is the query plan for the view definition, and q_{stmt} is the query plan for the main SQL statement. Within the main SQL statement, access to the `revenue0` view is done by using environment access as **Env**.`revenue0`. This approach cleanly handles views that rely on previously defined views, as well as dropping views.

In SQL, we also use the same approach to support `with-as` clauses. Generally speaking, NRA^e operations provide a natural way to represent let bindings within the query plans. In other words, NRA^e provides a simple way to represent shared sub-plans in the query and can naturally be used to handle SQL views, `with-as` clauses, or could be used to capture common sub-expression elimination rewrites.

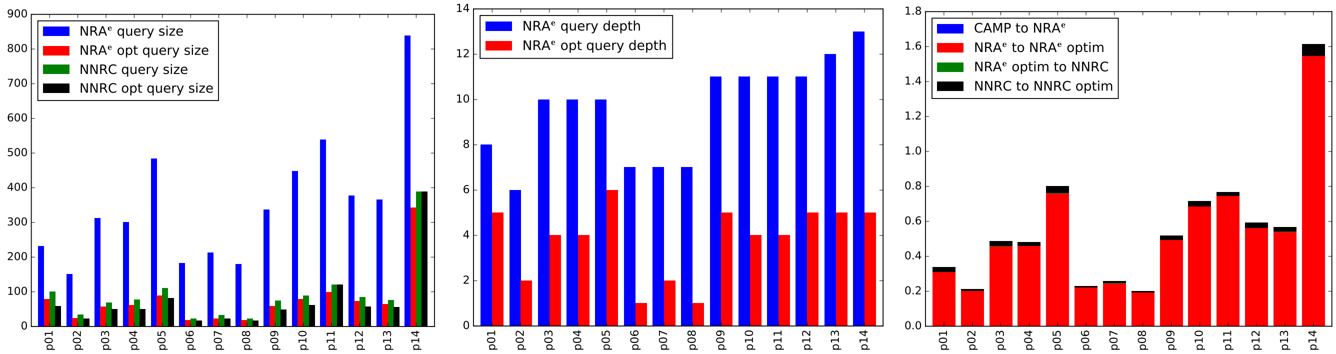
7. TRANSLATING RULES TO NRA^e

The second application is a translation from a query DSL of JRules [8] to NRA^e. It is the original motivation of the work [4]. While building a compiler for this language, we faced an explosion of the size of intermediate NRA queries. This section reports on how NRA^e overcomes this problem.

From CAMP to NRA^e. To illustrate how NRA^e simplifies the compilation of a production rule language, we use the Calculus for Aggregating Matching Patterns (CAMP) introduced in [34]. The syntax of the CAMP calculus ✿ is:

$$p ::= d \mid \boxplus p \mid p_1 \boxtimes p_2 \mid \textbf{it} \mid \textbf{env} \mid \textbf{let it} = p_1 \textbf{ in } p_2$$
$$\mid \textbf{let env} \mathrel{+}= p_1 \textbf{ in } p_2 \mid \textbf{map } p \mid \textbf{assert } p \mid p_1 \| p_2$$

We present here an intuitive semantics of CAMP; the formal semantics of the calculus is defined in [34] on the same data model we use for NRA^e. The language operates over an implicit datum being matched and an environment. The **it** construct obtains the implicit datum and **env** the environment. The **let it** $= p_1$ **in** p_2 construct uses the value of p_1 as

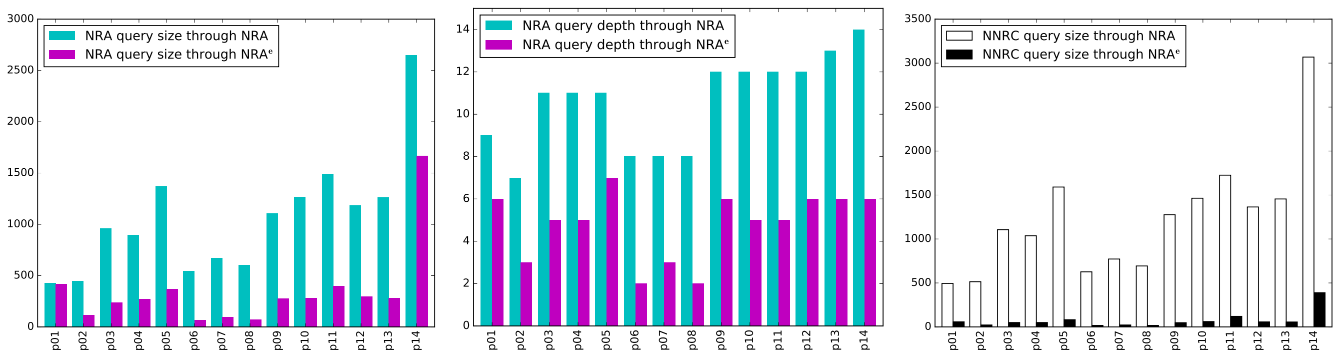


(a) NRA^e and NNRC query sizes.

(b) NRA^e query depths.

(c) Compilation time (s).

Figure 8: CAMP tests.



(a) NRA query sizes.

(b) NRA query depths.

(c) NNRC expression sizes.

Figure 9: Comparison of the direct translation from CAMP to NRA with translation through NRA^e.

implicit datum for p_2 and **let env** $+= p_1$ **in** p_2 updates the environment for the evaluation of p_2. The **map** p construct maps a pattern p over the implicit datum **it**. The **assert** p construct can introduce match failure and $p_1 \| p_2$ can recover from this kind of failure.

In [34], the translation from CAMP to NRA relies on two principles. (1) To encode the notion of recoverable errors, the output is always a bag. This bag is guaranteed to be either empty (representing a recoverable error) or a singleton of the datum. (2) The two inputs **it** and **env** of CAMP are encoded in the single input **In** of NRA. **In** is always a record with two fields, E and D, storing the environment and datum.

In translating from CAMP to NRA^e, we keep the first principle but eschew the second. Consider the translation of **it** and **env**. When we go from CAMP to NRA, we have to project the field corresponding to the value we want to access ($[\![p]\!]_r$ is the translation function of a pattern p):

$$[\![\mathbf{it}]\!]_r = \{\mathbf{In}.D\} \qquad [\![\mathbf{env}]\!]_r = \{\mathbf{In}.E\}$$

This result is wrapped in a bag to encode the pattern matching semantics. When going from CAMP to NRA^e, the two inputs of CAMP can be mapped to the two inputs of NRA^e. Hence, the translation of **it** and **env** becomes:

$$[\![\mathbf{it}]\!]_r = \{\mathbf{In}\} \qquad [\![\mathbf{env}]\!]_r = \{\mathbf{Env}\}$$

This direct mapping of the CAMP inputs to the NRA^e inputs simplifies the translation of other CAMP constructs. The complete translation function ✿ is given in Appendix A and has been proven correct ✿. This simplification is similar to the one observed in the examples with lambdas from Section 2. The ability to represent the environment operations directly in the algebra avoids having to encode them through nested queries. This allows the optimizer to simplify the query plan much more effectively.

Experiments. We report on experiments compiling several CAMP programs in Figure 8. The first test, `p01`, is the example given as Figure 6 in [34], `p02` is an example of select, `p03` is a join, `p04` and `p05` are joins with negation, `p06` to `p08` are simple aggregations, and `p09` to `p14` are joins with aggregation.

Figures 8a and 8b show the size and the depth of the intermediate queries when compiling a CAMP program to NNRC. Compared to the TPC-H benchmark queries, the NRA^e queries coming from CAMP programs have a similar size but they have a deeper level of nesting. The optimizer is much more effective on these queries than on the queries from the TPC-H benchmark. This is because CAMP programs were our primary goal and there are optimization tailored to remove translation artifacts from CAMP to NRA^e.

In term of compilation time (Figure 8c), compared to the TPC-H benchmark, the proportion spent in the NRA^e optimizer is higher than the one spent in the NNRC optimizer because more NRA^e optimizations are triggered and the NNRC terms to optimize are smaller. The compilation time remains on the order of a few seconds.

Figure 9 compares a direct translation from CAMP to NRA with a compilation path that goes through NRA^e. Figures 9a and 9b show for each of the examples the size and depth of NRA queries generated directly from CAMP or from CAMP to NRA^e and then to NRA. Figure 9c shows the size of the NNRC expressions generated by going through NRA and NRA^e. All the numbers are given after optimization.

The NRA^e optimizer includes two distinct categories of rewrites: (i) NRA^e rewrites like the ones presented in Section 4.3, and (ii) classic NRA rewrites lifted to NRA^e (some examples of rewrites are given Figure 12 in Appendix A). The second class of rewrites can thus also be applied on NRA queries.

A detailed analysis of the results shows that, for example, for `p01`, while none of the rewrites dedicated to NRA^e reduce the depth, they allow the pure NRA rewrites to "kick in", e.g., allowing the optimizer to recognize subqueries of the form $\chi_{\langle \mathbf{In} \rangle}(q)$ which can be removed by the simple rewrite: $\chi_{\langle \mathbf{In} \rangle}(q) \Rightarrow q$. As it turns out, this specific rewrite is never triggered when we optimize the NRA query coming directly from CAMP.

By comparing the NRA^e opt bars of Figure 8a with the through NRA^e bars of Figure 9a, we can see that even before optimization, the NRA^e queries are much smaller than the NRA queries. For example, `p01` goes from a size of 78 in NRA^e to 417 in NRA. This difference make the generated NNRC code much smaller (Figure 9c).

8. IMPLEMENTATION

The work presented in this paper is part of an effort in applying recent theorem proving technologies for the design, implementation and verification of query compilers. In this section, we review the state of our implementation and go over practical aspects that were omitted from the main part of the paper. We also report on our experience in using theorem proving technology for query compiler implementation.

*The Q*cert Compiler.* Figure 10 gives an overview of the full compiler architecture. Each square box corresponds to an intermediate representation specified using Coq. The red coloring identifies the subset of the compiler that is accompanied by mechanically checked correctness proofs. Those cover all parts described in this paper, except for the SQL to NRA^e translation. The compiler implementation is automatically generated from the mechanized specification using Coq's extraction mechanism [23], ensuring that the compiler matches the specification.

As of this writing, the implementation supports compilation from JRules (our initial target and the most complete), a fragment of SQL and OQL, and NRA$^\lambda$. It can emit code for execution in JavaScript or Java, for Spark, and for the Cloudant NoSQL database. In each case, the emitted code has to be linked with a small runtime library which implements core operations over the data model (e.g., record construction/access, collection operations such as flatten, distinct, etc). The Java and JavaScript backends are useful for testing and also serve as building blocks for the Spark and Cloudant backends. For Cloudant, the compiler produces map/reduce views containing JavaScript code.

From a front-end perspective, the system includes a parser for OQL and NRA$^\lambda$. JRules and SQL support rely on existing Java parsers for those languages, which pass an AST to the compiler encoded as an S-expression. The named nested relational calculus (NNRC) is the gateway to the backend

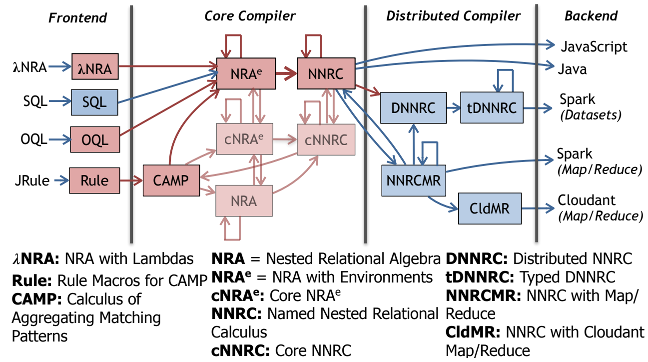


Figure 10: Q*cert compiler architecture.

part of the compiler. It is directly used to generate code for Java and JavaScript, while code generation for Spark and Cloudant rely on additional intermediate representations (DNNRC models distributed computation with Spark Datasets, NNRCMR models map/reduce, and CldMR models Cloudant-specific map/reduce).

Data model and type system. In order to focus on the novel features of NRA^e, this paper uses a simple data model of complex values with bags and records, and omits the treatment of types. This is far from enough for practical languages. For instance, OQL and JRules support class hierarchies, and all our languages require support for null values, date comparisons, and arithmetic. The implementation supports a rich data model that includes a notion of objects as well as sum types in a way similar to [21, 33]. The compiler specification and correctness proofs are done over that richer data model and type system. It is important to recall that type correctness is used pervasively as a pre-condition for algebraic rewrites and rely on type checking and type soundness proofs for the intermediate languages.

To handle additional data types (e.g., dates), and provide some modularity, the mechanization is parameterized over a notion of "foreign" types and operators. From a Coq perspective, those are *axioms* that are assumed by the proof system. A set of axioms for each foreign type typically includes semantics and typing judgments, with correctness properties ✿. When the compiler is extracted, an *implementation* of those axioms as regular code must be provided. Note that this choice represents a trade-off, since Coq cannot ensure the correctness of that part of the implementation.

Optimizer. Most of the formal treatment of the paper uses algebraic equivalences to convey optimizations. As we mentioned in the introduction, those are used as part of the proof of correctness for the optimizer itself. These optimizations are written as individual pattern-match based transformations (with side conditions as needed), each of which is proven to preserve semantics (and typing as appropriate).

The optimization infrastructure is parameterized by a list of rewrites and a cost function. All possible rewrites are applied through a depth-first AST traversal and optimization proceeds as long as the cost is decreasing. Despite being simple by traditional databases standards, the optimizer includes on the order of a hundred rewrites. The cost is currently based on the size and depth of the query which means there is a lot of room for improvements on that part of the implementation. Coq does not introduce specific limitations as to the complexity of the optimizer, which could use a search space and more complex cost model.

Finally, we have made some efforts to ensure that adding new optimizations is relatively straightforward. As shown in the introduction, each rewrite can be programmed and proved individually. The infrastructure provides a tactic that helps with this, in order to automatically reduce the full proof to a proof of the individual type-directed rewrite.

Support for other languages. Our current compiler uses well-known database representations which means it can easily be used to investigate or validate novel optimizations for other use cases, as we have done for CAMP. The amount of work required for adding a new language to the compiler depends on the nature of the language. Q*cert makes a few important assumptions: one is a data model based on nested relations, the other is that type checking is assumed in most of the optimizer support. As a result, the compiler should be a good match for Spark (which we partially support), Pig Latin [29], or JSON languages such as Jaql [7]. Additional work on typing for semi-structured data would make languages such as JSONiq [19] or SQL++ [30] interesting next steps. Languages such as XQuery would require significant changes to the compiler because of the complexity of the XML data model.

Thoughts on Coq. Coq is a functional programming language that fits well the task of writing compilers. Proofs can be added gradually, extraction is robust, and the code generated benefits from the good quality of the OCaml compiler, both in terms of stability and performances. The current performance bottlenecks in our implementation are in the optimizers, which can lead to an exponential number of iterations over the tree. However, it should be possible to use memoization techniques to improve performances [9].

One of the pleasant surprises of our experience has been the versatility of Coq, which we feel could be used to tackle a range of database problems. For instance, similar techniques could be used for: the specification of a query language after the fact or during development (for documentation and prototyping), ensuring the correctness of complex algorithms (e.g., view maintenance), or building an end-to-end verified compiler. This last scenario would certainly be a large undertaking and would require addressing integration issues with a real database engine.

9. RELATED WORK

There has been renewed interest in the formal verification of database systems or query languages [6, 11, 25, 26, 34]. So far, much of the work has focused on formalization [6, 11, 34] or on evaluating challenges involved in mechanization [26]. The closest related work is that of Cherniack and Zdonik [12, 13], which focused on the formal specification of rule-based query optimizers and used the Larch [20] theorem prover to verify correctness. Our work extends that approach in two ways: (i) we describe an alternative combinator-based algebra with built-in support for environments and (ii) we leverage recent advances in theorem proving technology to specify a much larger part of the query compiler.

How to best deal with variables and environments in algebraic compilers has received relatively little attention. For SPJ (Select-Project-Join) queries, variables can be eliminated at translation time and equivalences can be simply defined for a given static environment [2]. For query languages over complex or nested data, reification of the environment as a record is appealing in that existing relational techniques can be readily applied. This idea has notably been used in algebraic compilers for query languages over nested or graph data such as OQL [14] and XQuery [27, 32]. Full reification enables relational optimizations, but can result in large or highly nested plans in those languages as well. The algebra from [14] does combine environments and reification, but assumes that environments are fixed for the purpose of defining plan equivalence.

Dealing with bindings is also important for the formalization of programming languages. The POPLmark challenge [5] has helped spur an assortment of techniques for representing and reasoning about bindings. These are all focused on traditional bindings, as introduced by functions. Our work uses explicit reified environments instead. It enables support for the standard shadowing semantics for occurrences of a variable while also supporting unification semantics, where the value of the variable added to the environment has to be compatible with previous occurrences.

10. CONCLUSION

This paper introduced NRA^e, the Nested Relational Algebra with Environments, which provides the foundation for a formally verified query compiler written using the Coq proof assistant. NRA^e extends a combinators-based nested relational algebra with explicit environment support in a way that facilitates the specification and verification of algebraic rewrites. A lifting theorem shows that all existing NRA optimizations also apply to NRA^e. We showed how the resulting compiler can be used for both traditional queries and for a query DSL in the context of a rules language.

Theorem proving techniques have greatly matured in recent years, and we feel that their application to databases could prove useful in a number of ways (for specification, prototyping, or to provide correctness guarantees in scenarios where security or privacy are important). We are currently working on further improvements to our compiler infrastructure, notably to the optimizer and backend, in order to support code-generation for distributed query plans.

Acknowledgments. We would like to thank the anonymous reviewers for their comments and suggestions which greatly helped us improve the content and presentation of this work. We also thank Guillaume Baudart, Stefan Fehrenbach, and Erik Wittern for their feedback on earlier drafts. Finally, we send this ✿ to Florence Plateau.

11. REFERENCES

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming (JFP)*, 1(04):375–416, 1991.

[2] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems (TODS)*, 4(4):435–454, 1979.

[3] M. Armbrust et al. Spark SQL: Relational data processing in Spark. In *International Conference on*

Management of Data (SIGMOD), pages 1383–1394, 2015.

[4] M. Arnold et al. META: Middleware for events, transactions, and analytics. *IBM Journal of Research and Development (IBMRD)*, 60(2-3), 2016.

[5] B. E. Aydemir et al. Mechanized metatheory for the masses: The PoplMark challenge. In *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 50–65, 2005.

[6] V. Benzaken, E. Contejean, and S. Dumbrava. A Coq formalization of the relational data model. In *European Symposium on Programming (ESOP)*, pages 189–208, 2014.

[7] K. Beyer et al. JAQL: A scripting language for large scale semistructured data analysis. In *Conference on Very Large Data Bases (VLDB)*, pages 1272–1283, 2011.

[8] J. Boyer and H. Mili. *Agile Business Rule Development*. Springer, 2011.

[9] T. Braibant, J.-H. Jourdan, and D. Monniaux. Implementing and reasoning about hash-consed data structures in Coq. *Journal of Automated Reasoning*, 53(3):271–304, 2014.

[10] M. J. Carey et al. The architecture of the EXODUS extensible DBMS. In *On Object-Oriented Database Systems*, pages 231–256. Springer Berlin Heidelberg, 1991.

[11] J. Cheney and C. Urban. Mechanizing the metatheory of mini-XQuery. In *Conference on Certified Programs and Proofs (CPP)*, pages 280–295, 2011.

[12] M. Cherniack and S. B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *International Conference on Management of Data (SIGMOD)*, 1996.

[13] M. Cherniack and S. B. Zdonik. Changing the rules: Transformations for rule-based optimizers. In *International Conference on Management of Data (SIGMOD)*, pages 61–72, 1998.

[14] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Workshop on Database Programming Languages (DBPL)*, pages 226–242, 1993.

[15] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296, 2007.

[16] The Coq proof assistant reference manual, version 8.4pl41. coq.inria.fr.

[17] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000.

[18] L. Fegaras, D. Maier, and T. Sheard. Specifying rule-based query optimizers in a reflective framework. In *Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 146–168, 1993.

[19] D. Florescu and G. Fourny. JSONiq: The history of a query language. *IEEE Internet Computing*, 17(5):86–90, 2013.

[20] S. J. Garland and J. V. Guttag. An overview of LP, the Larch Prover. In *Conference on Rewriting Techniques and Applications (RTA)*, pages 137–151, 1989. nms.lcs.mit.edu/Larch.

[21] G. Giorgidze, T. Grust, A. Ulrich, and J. Weijers. Algebraic data types for language-integrated queries. In *Workshop on Data Driven Functional Programming (DDFP)*, pages 5–10, 2013.

[22] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-supported program execution. In *International Conference on Management of Data (SIGMOD)*, pages 1063–1066, 2009.

[23] P. Letouzey. A new extraction for Coq. In *Conference on Types for Proofs and Programs (TYPES)*, pages 200–219, 2003.

[24] T. W. Leung et al. The AQUA data model and algebra. In *Workshop on Database Programming Languages (DBPL)*, pages 157–175. 1994.

[25] G. Malecha and R. Wisnesky. Using dependent types and tactics to enable semantic optimization of language-integrated queries. In *Symposium on Database Programming Languages (DBPL)*, pages 49–58, 2015.

[26] J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Principles of Programming Languages (POPL)*, 2010.

[27] N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *ACM Transactions on Database Systems (TODS)*, 31(3):968–1013, 2006.

[28] E. Meijer. The world according to LINQ. *Communications of the ACM (CACM)*, 54(10):45–51, 2011.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *International Conference on Management of Data (SIGMOD)*, pages 1099–1110, 2008.

[30] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. *CoRR*, abs/1405.3631, 2014.

[31] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *International Conference on Management of Data (SIGMOD)*, pages 39–48, 1992.

[32] C. Ré, J. Siméon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *International Conference on Data Engineering (ICDE)*, 2006.

[33] A. Shinnar and J. Siméon. A branding strategy for business types. In *Wadlerfest'2016*, Edinburgh, Scotland, April 2016. LNCS.

[34] A. Shinnar, J. Siméon, and M. Hirzel. A pattern calculus for rule languages: Expressiveness, compilation, and mechanization. In *European Conference for Object-Oriented Programming (ECOOP)*, pages 542–567, 2015.

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

APPENDIX

A. CAMP

Section 7 discussed the CAMP calculus, which was originally described in [34] as a useful intermediate language for compiling rule-based languages. Section 7 reported on the results of translating and optimizing CAMP through NRA^e. In this appendix, we formalize the translation from CAMP to NRA^e, and present some important NRA optimizations that help simplify the resulting NRA^e.

The paper that introduced the CAMP calculus presents a translation into NRA in Figure 10 of [34]. This required encoding the input as a record with two components D and E representing the current data and environment. NRA^e avoids the need for such an encoding, as the CAMP environment can be directly represented using the NRA^e environment. Figure 11 presents the full translation from CAMP to NRA^e (on the right). For comparison, the original translation from [34] is presented in parallel (on the left).

Consider for example the translation of **map** p. When we go to NRA^e, the translation produces a corresponding map in NRA^e, and uses a flattening to account for the fact that the result of translating p will return a collection:

$$[\![\mathbf{map}\ p]\!]_r = \{\mathit{flatten}\left(\chi_{\langle [\![p]\!]\rangle}(\mathbf{In})\right)\}$$

When we go to NRA, in addition to the flattening, the input must be manipulated to iterate on the data part and keep the environment. The translation function is:

$$[\![\mathbf{map}\ p]\!]_r = \\ \{\mathit{flatten}\left(\chi_{\langle [\![p]\!]\rangle}\left(\rho_{D/\{T\}}\left(\{[E:\mathbf{In}.E]\oplus[T:\mathbf{In}.D]\}\right)\right)\right)\}$$

where $\rho_{B/\{A\}}\,(q)$ is the unnest operator from Section 3.2.

In addition to the NRA^e specific optimizations presented in Figure 3, many standard NRA optimizations are useful for optimizing translated CAMP. Figure 12 presents a number of these optimizations, which serve to eliminate inefficiencies introduced either by the structure of the CAMP language or naive translation. Similarly, Figure 13 presents more complex NRA^e optimizations that target common patterns produced by compilation from CAMP.

NRA		CAMP		NRA^e
$\{d\}$	$=$	$\llbracket d \rrbracket_r$	$=$	$\{d\}$
$\chi_{\langle \boxplus \mathbf{In} \rangle}(\llbracket p \rrbracket_r)$	$=$	$\llbracket \boxplus p \rrbracket_r$	$=$	$\chi_{\langle \oplus \mathbf{In} \rangle}(\llbracket p \rrbracket_r)$
$\Big(\chi_{\langle \mathbf{In}.T_1 \boxtimes \mathbf{In}.T_2 \rangle} \quad \chi_{\langle [T_1:\mathbf{In}] \rangle}(\llbracket p_1 \rrbracket) \times \chi_{\langle [T_2:\mathbf{In}] \rangle}(\llbracket p_2 \rrbracket)\Big)$	$=$	$\llbracket p_1 \boxtimes p_2 \rrbracket_r$	$=$	$\Big(\chi_{\langle \mathbf{In}.T_1 \boxtimes \mathbf{In}.T_2 \rangle} \quad \chi_{\langle [T_1:\mathbf{In}] \rangle}(\llbracket p_1 \rrbracket) \times \chi_{\langle [T_2:\mathbf{In}] \rangle}(\llbracket p_2 \rrbracket)\Big)$
$\Big\{ \mathit{flatten}\Big(\chi_{\langle \llbracket p \rrbracket \rangle} \quad \big(\rho_{D/\{T\}}\left(\{[E:\mathbf{In}.E] * [T:\mathbf{In}.D]\}\right)\big)\Big)\Big\}$	$=$	$\llbracket \mathbf{map}\ p \rrbracket_r$	$=$	$\big\{\mathit{flatten}\left(\chi_{\langle \llbracket p \rrbracket \rangle}(\mathbf{In})\right)\big\}$
$\chi_{\langle [\,] \rangle}\left(\sigma_{\langle \mathbf{In} \rangle}(\llbracket p \rrbracket_r)\right)$	$=$	$\llbracket \mathbf{assert}\ p \rrbracket_r$	$=$	$\chi_{\langle [\,] \rangle}\left(\sigma_{\langle \mathbf{In} \rangle}(\llbracket p \rrbracket_r)\right)$
$\llbracket p_1 \rrbracket \Vert \llbracket p_2 \rrbracket$	$=$	$\llbracket p_1 \Vert p_2 \rrbracket_r$	$=$	$\llbracket p_1 \rrbracket_r \Vert \llbracket p_2 \rrbracket_r$
$\{\mathbf{In}.D\}$	$=$	$\llbracket \mathbf{it} \rrbracket_r$	$=$	$\{\mathbf{In}\}$
$\mathit{flatten}\Big(\chi_{\langle \llbracket p_2 \rrbracket_r \rangle} \quad \big(\rho_{D/\{T\}}\left(\{[E:\mathbf{In}.E] * [T:\llbracket p_1 \rrbracket_r]\}\right)\big)\Big)$	$=$	$\llbracket \mathbf{let\ it} = p_1\ \mathbf{in}\ p_2 \rrbracket_r$	$=$	$\mathit{flatten}\left(\chi_{\langle \llbracket p_2 \rrbracket \rangle}(\llbracket p_1 \rrbracket_r)\right)$
$\{\mathbf{In}.E\}$	$=$	$\llbracket \mathbf{env} \rrbracket_r$	$=$	$\{\mathbf{Env}\}$
$\mathit{flatten}\Big(\chi_{\langle \llbracket p_2 \rrbracket_r \rangle}\big(\chi_{\langle [E:\mathbf{In}.E_2] * [D:\mathbf{In}.D] \rangle}\big(\rho_{E_2/\{T_2\}}\big(\chi_{\langle \mathbf{In} * [T_2:\mathbf{In}.E + \mathbf{In}.E_1] \rangle}\big(\rho_{E_1/\{T_1\}}\left(\{\mathbf{In} * [T_1 : \llbracket p_1 \rrbracket_r]\}\right)\big)\big)\big)\big)\Big)$	$=$	$\llbracket \mathbf{let\ env} \mathrel{+}= p_1\ \mathbf{in}\ p_2 \rrbracket_r$	$=$	$\mathit{flatten}\Big(\chi^e_{\langle \llbracket p_2 \rrbracket \rangle} \circ^e \mathit{flatten}\left(\chi_{\langle \mathbf{In} \otimes \mathbf{Env} \rangle}(\llbracket p_1 \rrbracket)\right)\Big)$

Figure 11: From CAMP to NRA ✿ and NRA^e ✿ $\boxed{\llbracket p \rrbracket_r = q}$

$$
\begin{aligned}
[a:q].a &\Rightarrow q & ✿\\
(q_1 \oplus [a_2:q_2]).a_2 &\Rightarrow q_2 & ✿\\
\text{if } a_1 \neq a_2,\ (q \oplus [a_2:q_2]).a_1 &\Rightarrow q.a_1 & ✿\\
\text{if } a_1 \neq a_2,\ ([a_1:q_1] \oplus q).a_2 &\Rightarrow q.a_2 & ✿\\
[\,] \otimes q &\Rightarrow \{q\} & ✿\\
q \otimes [\,] &\Rightarrow \{q\} & ✿\\
\{[a_1:q_1]\} \times \{[a_2:q_2]\} &\Rightarrow \{[a_1:q_1] \oplus [a_2:q_2]\} & ✿\\
\mathbf{In} \circ q &\Rightarrow q & ✿\\
(\boxplus(q_1)) \circ q_2 &\Rightarrow \boxplus(q_1 \circ q_2) & ✿\\
(q_2 \boxtimes q_1) \circ q &\Rightarrow (q_2 \circ q) \boxtimes (q_1 \circ q) & ✿\\
\text{if } \mathcal{I}^i(q_1),\ q_1 \circ q_2 &\Rightarrow q_1 & ✿
\end{aligned}
$$

$$
\begin{aligned}
\chi_{\langle q_1 \rangle}(q_2) \circ q &\Rightarrow \chi_{\langle q_1 \rangle}(q_2 \circ q) & ✿\\
\mathit{flatten}(\chi_{\langle \chi_{\langle \{q_3\} \rangle}(q_1) \rangle}(q_2)) &\Rightarrow \chi_{\langle \{q_3\} \rangle}\left(\mathit{flatten}(\chi_{\langle q_1 \rangle}(q_2))\right) & ✿\\
\chi_{\langle p_1 \rangle}\left(\mathit{flatten}(p_2)\right) &\Rightarrow \mathit{flatten}(\chi_{\langle \chi_{\langle p_1 \rangle}(\mathbf{In}) \rangle}(p_2)) & ✿\\
\chi_{\langle p_1 \rangle}\left(\mathit{flatten}(\chi_{\langle p_2 \rangle}(p_3))\right) &\Rightarrow \mathit{flatten}(\chi_{\langle \chi_{\langle p_1 \rangle}(p_2) \rangle}(p_3)) & ✿\\
\mathit{flatten}(\{q\}) &\Rightarrow q & ✿\\
\mathit{flatten}(\chi_{\langle \{q_1\} \rangle}(q_2)) &\Rightarrow \chi_{\langle q_1 \rangle}(q_2) & ✿\\
\chi_{\langle \mathbf{In} \rangle}(q) &\Rightarrow q & ✿\\
\chi_{\langle q_1 \rangle}\left(\chi_{\langle q_2 \rangle}(q)\right) &\Rightarrow \chi_{\langle q_1 \circ q_2 \rangle}(q) & ✿\\
\chi_{\langle q_1 \rangle}(\{q_2\}) &\Rightarrow \{q_1 \circ q_2\} & ✿\\
\chi_{\langle q_2 \rangle}\left(\sigma_{\langle q_1 \rangle}(\{q\})\right) &\Rightarrow \chi_{\langle q_2 \circ q \rangle}\left(\sigma_{\langle q_1 \circ q \rangle}(\{\mathbf{In}\})\right) & ✿
\end{aligned}
$$

Figure 12: NRA rewrites for CAMP.

$$
\begin{aligned}
\mathit{flatten}(\chi^e_{\langle \chi_{\langle \mathbf{Env} \rangle}(\sigma_{\langle q_1 \rangle}(\{\mathbf{In}\})) \rangle}) \circ^e \chi_{\langle \mathbf{Env} \rangle}\left(\sigma_{\langle q_2 \rangle}(\{\mathbf{In}\})\right) &\Rightarrow \chi_{\langle \mathbf{Env} \rangle}\left(\sigma_{\langle q_1 \rangle}\left(\sigma_{\langle q_2 \rangle}(\{\mathbf{In}\})\right)\right) & ✿\\
(\chi^e_{\langle q \rangle}) \circ^e (\mathbf{Env} \otimes [a:\mathbf{In}]) &\Rightarrow \chi_{\langle (q \circ \mathbf{Env}.a) \circ^e \mathbf{In} \rangle}(\mathbf{Env} \otimes [a:\mathbf{In}]) & ✿\\
\mathit{flatten}(\chi^e_{\langle q \rangle}) \circ^e (\mathbf{Env} \otimes [a:\mathbf{In}]) &\Rightarrow \mathit{flatten}(\chi_{\langle (q \circ \mathbf{Env}.a) \circ^e \mathbf{In} \rangle}(\mathbf{Env} \otimes [a:\mathbf{In}])) & ✿\\
\chi_{\langle \mathbf{Env} \otimes \mathbf{In} \rangle}\left(\sigma_{\langle q_1 \rangle}(\mathbf{Env} \otimes q_2)\right) &\Rightarrow \chi_{\langle \{\mathbf{In}\} \rangle}\left(\sigma_{\langle q_1 \rangle}(\mathbf{Env} \otimes q_2)\right) & ✿
\end{aligned}
$$

Figure 13: NRA^e rewrites for CAMP.