Using Dependent Types and Tactics to Enable Semantic Optimization of Language-Integrated Queries

Gregory Malecha
University of California, San Diego
gmalecha@eng.ucsd.edu

Ryan Wisnesky [*]
Massachusetts Institute of Technology
wisnesky@math.mit.edu

Abstract

Semantic optimization — the use of data integrity constraints to optimize relational queries — has been well studied but, owing to limitations in how SQL handles constraints, has not often been applied by mainstream RDBMSs. In a language-integrated query setting, however, the query provider is free to rewrite queries before they are executed on an RDBMS. We show, using Coq as our ambient language, how to use dependent types to represent a well known class of constraints — embedded, implicational dependencies — and how Coq tactics can be used to implement a particular kind of semantic optimization: tableaux minimization, which minimizes the number of joins required by a query.

1. Introduction

Semantic optimization [1, 8, 19] is the use of data integrity constraints such as keys, functional dependencies, inclusions, and join decompositions to optimize relational queries. For example [1], consider the following contrived query over a relation (set of records) $Movies$ with fields title, director, and actor:

$$\begin{aligned}&\texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies)\\ &\texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title}\\ &\texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})\end{aligned}$$

This query returns (a set of) tuples (d, a) where a acted in a movie directed by d. A naïve implementation of this query will require a join. However, when $Movies$ satisfies the functional dependency $\mathsf{title} \to \mathsf{director}$ (meaning that if $(\mathsf{director} : d, \mathsf{title} : t, \mathsf{actor} : a)$ and $(\mathsf{director} : d', \mathsf{title} : t', \mathsf{actor} : a')$ are $Movies$ records such that $t = t'$, then $d = d'$), this query is equivalent to:

$$\begin{aligned}&\texttt{for}\ (m\ \texttt{in}\ Movies)\\ &\texttt{return}\ (m.\mathsf{director}, m.\mathsf{actor})\end{aligned}$$

which can be evaluated without a join. (Note that if $Movies$ did not satisfy the functional dependency, the equivalence would not necessarily hold.)

Of course, knowing that the functional dependency holds, a programmer might simply write the optimized query to begin with. However, constraints are not always known at compile time, such as when relations are indexed dynamically. In addition, not all queries are written by programmers. For example, information-integration systems such as Clio [14] automatically generate large numbers of un-optimized queries.

Although certain RDBMS's such as DB2 can perform limited amounts of semantic optimization [15], RDBMS's are fundamentally limited by the expressiveness of SQL as a constraint specification language. For example, SQL includes keys and foreign keys but constraints such as the functional dependency above are not directly expressible in SQL. (Technically, functional dependencies can be encoded as `CHECK` constraints, but even `CHECK` constraints cannot capture multi-table constraints such as join decompositions.) In relational database theory, a fragment of first-order logic, *embedded, implicational dependencies* (EDs) are used to capture almost all constraints used in practice, including keys, foreign keys, inclusions, functional dependencies, and join decompositions. A large body of literature has been developed to facilitate reasoning about queries in the presence of EDs [19].

Contributions and Outline In this paper we demonstrate that dependently-typed language-integrated query systems (LINQs [13]) that compile to SQL can expose data integrity constraints, in the guise of EDs, as first-class objects to their users. This feature allows them to apply sophisticated semantic optimization techniques before translating user queries into SQL. In particular, we show, using Coq [5] as our ambient language, how to use dependent equality types to represent EDs, and how to use Coq tactics to implement a particular kind of semantic optimization: tableaux minimization, which minimizes the number of joins required by a query. This paper is divided into two parts: the first part is a tutorial on tableaux minimization, and the second part is a Coq rendering of the first part. The Coq development is available at github.com/gmalecha/semantic-query.

Related Work Most theoretical work on language-integrated query systems is done in a simply-typed setting [21]. In practice, however, sophisticated type systems are often used to to facilitate embedding the query sublanguage into a general purpose programming language. For example, labeled row types [11] can be used to embed DBMS records into a programming language, and the Opaleye library for Haskell uses the Arrow type-class to statically enforce the well-formedness of its SQL output [10]. Rarer still are dependently-typed embedded query languages. Although Coq has been used to prove the correctness of certain database-related languages, data structures, and algorithms [4, 18], none of this work is concerned with using Coq directly as an embedded query language as we are doing in this paper (i.e., these works use deep embeddings of query languages, while we use a shallow embedding).

[*] Work supported by ONR grant N000141310260 and AFOSR grant FA9550-14-1-0031

[To appear in DBPL 2015.]

2. Queries

In this paper we will focus on relational *conjunctive queries* [1], and for the first part of this paper the specifics of our query language will not matter. We will write $(l_1 : e_1, \ldots, l_N : e_N)$ to indicate a record with unique labels $l_1, \ldots l_N$ formed from expressions $e_1, \ldots, e_N$, where an expression has the form $v.l$ for a variable v and label l. We will abbreviate (potentially 0-length) vectors of variables $x_1, ..., x_N$ as $\overrightarrow{x}$. We will write $P(\overrightarrow{x})$ to indicate a conjunction of equalities over expressions over variables $\overrightarrow{x}$. Assumed base relations (often called *roots*) will be written in capital letters, such as $\overrightarrow{X}$. A *tableau* has the form:

$$\begin{array}{l} \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ X)} \\ \texttt{where}\ P(\overrightarrow{x}) \end{array}$$

The $\overrightarrow{(x\ \texttt{in}\ X)}$ are called *generators*. A (conjunctive) *query* is a pair of a tableau and a record ("return clause") $R(\overrightarrow{x})$:

$$\begin{array}{l} \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ X)} \\ \texttt{where}\ P(\overrightarrow{x}) \\ \texttt{return}\ R(\overrightarrow{x}) \end{array}$$

Extensions We will only consider relational conjunctive queries in this paper, but many extensions to conjunctive queries have been studied in the literature [1]. Three extensions are particularly important, because many results about semantic optimization, including tableaux minimization, hold for these extensions [19]:

- It is possible to allow generators to be dependent, thereby allowing, for example, nested relations [19]:

  $$\texttt{for}\ (g\ \texttt{in}\ \mathit{Groups})\ (p\ \texttt{in}\ g)\ \ldots$$

- It is possible to interpret queries in arbitrary *monads with zeroes*, for example, the list monad or the bag monad. However, the optimization procedure described in this paper is only sound for monads that are both commutative and idempotent [19]:

  $$\texttt{for}\ (x\ \texttt{in}\ X)(y\ \texttt{in}\ Y) \cong \texttt{for}\ (y\ \texttt{in}\ Y)(x\ \texttt{in}\ X)$$

  $$\texttt{for}\ (x\ \texttt{in}\ X) \cong \texttt{for}\ (x\ \texttt{in}\ X)(x\ \texttt{in}\ X)$$

  Such monads arise, for example, as power monads on topoi [3].

- It is also possible to interpret queries in *monad algebras* [16]. For example, it is possible to write a query to find the largest element of a set:

  $$\texttt{for}\ (x\ \texttt{in}\ \mathit{SetOfInts})\ \texttt{max}\ x$$

3. Embedded Dependencies

An *embedded dependency (ED)* [1] is a pair of tableaux, where one tableau is universally quantified, and the other tableau is existentially quantified:

$$\begin{array}{lll} C & := & \texttt{forall}\ \overrightarrow{(x\ \texttt{in}\ X)} \\ & & \texttt{where}\ P(\overrightarrow{x}) \\ & & \texttt{exists}\ \overrightarrow{(y\ \texttt{in}\ Y)} \\ & & \texttt{where}\ B(\overrightarrow{x}, \overrightarrow{y}) \end{array}$$

Example The functional dependency from our example from the introduction is written (the `exists` clause is empty):

$$\begin{array}{l} \texttt{forall}\ (x\ \texttt{in}\ \mathit{Movies})\ (y\ \texttt{in}\ \mathit{Movies}) \\ \texttt{where}\ x.\mathsf{title} = y.\mathsf{title}, \\ \texttt{exists} \\ \texttt{where}\ x.\mathsf{director} = y.\mathsf{director} \end{array}$$

An ED C gives rise to two conjunctive queries, the *front* and *back* of C. We write $\mathcal{L}(\overrightarrow{x})$ to indicate a record capturing the variables $\overrightarrow{x}$; e.g., $(\mathsf{x_1} : x_1, \ldots, \mathsf{x_N} : x_N)$.

$$\begin{array}{lll} \mathit{front}(C) & := & \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ X)} \\ & & \texttt{where}\ P(\overrightarrow{x}) \\ & & \texttt{return}\ \mathcal{L}(\overrightarrow{x}) \end{array}$$

$$\begin{array}{lll} \mathit{back}(C) & := & \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ X)}\ \overrightarrow{(y\ \texttt{in}\ Y)} \\ & & \texttt{where}\ P(\overrightarrow{x}) \wedge B(\overrightarrow{x}, \overrightarrow{y}) \\ & & \texttt{return}\ \mathcal{L}(\overrightarrow{x}) \end{array}$$

It is easy to establish [19] that

$$\forall I, \quad I \models C \quad \text{iff} \quad \mathit{front}(C)(I) = \mathit{back}(C)(I)$$

In the above, $I \models C$ should be read "constraint C holds on instance I." In the second half of this paper, we will use a dependent equality type corresponding to the above equation as a type of proofs that an ED holds in a particular instance.

Notation When two queries Q_1 and Q_2 give the same result on every instance, we write $Q_1 \cong Q_2$. When Q_1 and Q_2 give the same result on every instance satisfying some set of EDs C, we write $C \vdash Q_1 \cong Q_2$.

4. Homomorphisms

A *homomorphism* $h : Q_1 \to Q_2$ between queries:

$$\begin{array}{lcl} \texttt{for}\ \overrightarrow{(v_1\ \texttt{in}\ V_1)} & & \texttt{for}\ \overrightarrow{(v_2\ \texttt{in}\ V_2)} \\ \texttt{where}\ P_1(\overrightarrow{v_1}) & \to_h & \texttt{where}\ P_2(\overrightarrow{v_2}) \\ \texttt{return}\ R_1(\overrightarrow{v_1}) & & \texttt{return}\ R_2(\overrightarrow{v_2}) \end{array}$$

is a substitution mapping the `for` -bound variables of Q_1 (namely, $\overrightarrow{v_1}$) to the `for` -bound variables of Q_2 (namely, $\overrightarrow{v_2}$) that preserves the structure of Q_1 in the sense that:

- $(h(v_{1_i})\ \texttt{in}\ V_{1_i}) \in \overrightarrow{(v_2\ \texttt{in}\ V_2)}$ – the image of each generator in Q_1 is found in the generators of Q_2.
- $P_2(\overrightarrow{v_2}) \vdash P_1(h(\overrightarrow{v_1}))$ – the image of the where clause of Q_1 is entailed by the where clause of Q_2.
- $P_2 \vdash R_1(h(\overrightarrow{v_1})) = R_2(\overrightarrow{v_2})$ – the image of the return clause of Q_1 is equal to the return clause of Q_2, under P_2.

A homomorphism of tableaux is defined the same way, except that the condition about `return` clauses is dropped.

Notation We write $Q_1 \leftrightarrow Q_2$ to mean that there exists homomorphisms $Q_1 \to Q_2$ and $Q_2 \to Q_1$ and we say that Q_1 and Q_2 are *homomorphically equivalent*. The existence of a homomorphism $Q_1 \to Q_2$ implies that for every I, $Q_2(I) \subseteq Q_1(I)$, and vice versa [1]. Hence, by bi-directional subset containment, $Q_1 \cong Q_2$ iff $Q_1 \leftrightarrow Q_2$.

Example Consider our *Movies* query (Q_1) and its semantically optimized counter-part (Q_2):

$$\begin{array}{lll} Q_1 & := & \texttt{for}\ (m_1\ \texttt{in}\ \mathit{Movies})\ (m_2\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \\ & & \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \\ Q_2 & := & \texttt{for}\ (m\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{return}\ (m.\mathsf{director}, m.\mathsf{actor}) \end{array}$$

It is easy to see that for every instance I that satisfies the embedded dependency above, $Q_2(I) \subseteq Q_1(I)$. To check that there is a homomorphism $h : Q_1 \to Q_2$; namely, the substitution $m_1 \mapsto m, m_2 \mapsto m$, we first apply h to Q_1 (which will "shadow" the name m):

$$\begin{array}{lll} h(Q_1) & := & \texttt{for}\ (m\ \texttt{in}\ \mathit{Movies})\ (m\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{where}\ m.\mathsf{title} = m.\mathsf{title} \\ & & \texttt{return}\ (m.\mathsf{director}, m.\mathsf{actor}) \end{array}$$

We see that each generator $(m\ \texttt{in}\ \mathit{Movies})$ in $h(Q_1)$ appears in Q_2. Next, we see that the `where` clause of $h(Q_1)$ is a tautology and hence is entailed by the (empty) `where` clause of Q_2. Finally, we see that the two `return` clauses are equal, and conclude that $m_1 \mapsto m, m_2 \mapsto m$ is a homomorphism.

Note that there is no homomorphism $Q_2 \to Q_1$, and so $Q_1 \not\cong Q_2$. There are only two candidates: $m \mapsto m_1$ and $m \mapsto m_2$. Neither works since the image of Q_2's `return` clause under either substitution (i.e. either `return` $(m_1.\mathsf{director}, m_1.\mathsf{actor})$ or `return` $(m_2.\mathsf{director}, m_2.\mathsf{actor})$) is equivalent to Q_1's `return` clause (`return` $(m_1.\mathsf{director}, m_2.\mathsf{actor})$) under the equality in Q_1 $(m_1.\mathsf{title} = m_2.\mathsf{title})$.

5. The Chase

The chase is a confluent rewriting procedure that rewrites queries using EDs [1]. Let

$$\begin{array}{lll} C & := & \texttt{forall}\ \overrightarrow{(x\ \texttt{in}\ X)} \\ & & \texttt{where}\ P(\overrightarrow{x}) \\ & & \texttt{exists}\ \overrightarrow{(y\ \texttt{in}\ Y)} \\ & & \texttt{where}\ B(\overrightarrow{x}, \overrightarrow{y}) \end{array} \qquad \begin{array}{lll} Q & := & \texttt{for}\ \overrightarrow{(v\ \texttt{in}\ V)} \\ & & \texttt{where}\ O(\overrightarrow{v}) \\ & & \texttt{return}\ R(\overrightarrow{v}) \end{array}$$

and suppose there exists a (tableau) homomorphism $h : \mathit{front}(C) \to Q$. A *chase step* rewrites Q into $\mathit{step}(C, Q)$ by adding the image of the back of C:

$$\begin{array}{lll} \mathit{step}(C, Q) & := & \texttt{for}\ \overrightarrow{(v\ \texttt{in}\ V)}\ \overrightarrow{(y\ \texttt{in}\ Y)} \\ & & \texttt{where}\ O(\overrightarrow{v}) \wedge B(\overrightarrow{h(x)}, \overrightarrow{y}) \\ & & \texttt{return}\ R(\overrightarrow{v}) \end{array}$$

A chase step is semantics-preserving on instances that satisfy the constraints [19], i.e.

$$C \vdash Q \cong \mathit{step}(C, Q)$$

The *chase* algorithm itself simply repeats the chase step until it finds a fixed point (up to homomorphic equivalence). That is:

$$\begin{array}{l} \mathit{chase}(C, Q) \equiv Q' \quad \text{iff} \\ \quad Q \rightsquigarrow \mathit{step}(C, Q) \rightsquigarrow \mathit{step}(C, \mathit{step}(C, Q)) \rightsquigarrow \ldots \rightsquigarrow Q' \end{array}$$

Termination of the chase is undecidable, but if it terminates the final result is unique (up to homomorphic equivalence) [8]. Provided certain fairness conditions are met [8], the chase extends easily to sets of EDs by choosing a particular ED to chase with at each step.

A key theorem about the chase is that it reduces the question of query equivalence under constraints to homomorphic equivalence. This means that, if $\vec{C}$ is a set of EDs and Q_1 and Q_2 are queries:

$$\vec{C} \vdash Q_1 \cong Q_2 \quad \text{iff} \quad \mathit{chase}(\vec{C}, Q_1) \leftrightarrow \mathit{chase}(\vec{C}, Q_2)$$

Example As we showed in the previous example, there is a homomorphism $x \mapsto m_1, y \mapsto m_2$ from the front of our constraint

$$\begin{array}{lll} C & := & \texttt{forall}\ (x\ \texttt{in}\ \mathit{Movies})\ (y\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{where}\ x.\mathsf{title} = y.\mathsf{title}, \\ & & \texttt{exists} \\ & & \texttt{where}\ x.\mathsf{director} = y.\mathsf{director} \end{array}$$

to our original query:

$$\begin{array}{lll} Q_1 & := & \texttt{for}\ (m_1\ \texttt{in}\ \mathit{Movies})\ (m_2\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \\ & & \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \end{array}$$

Hence, we can take a chase step:

$$\begin{array}{lll} \mathit{step}(C, Q_1) & := & \texttt{for}\ (m_1\ \texttt{in}\ \mathit{Movies})\ (m_2\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \wedge \\ & & \qquad m_1.\mathsf{director} = m_2.\mathsf{director} \\ & & \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \end{array}$$

which adds the `where` clause of the back of the constraint to the query's `where` clause. At this point we stop chasing, since $\mathit{step}(C, \mathit{step}(C, Q_1))$ is homomorphically equivalent (even syntactically equivalent) to $\mathit{step}(C, Q_1)$. By the soundness of the chase we have established that $C \vdash Q_1 \cong \mathit{chase}(C, Q_1)$.

6. Tableaux Minimization

We now demonstrate how to minimize queries in the presence of EDs, using a technique known as "tableaux minimization using chase and back-chase" [8]. Suppose we are given a query Q and set of EDs C. We first chase Q with C to obtain U, a so-called *universal plan*. We then search for sub-queries of U (which are intuitively obtained by removing generators from U), chasing each in turn with C to check for equivalence with U. There will always be a unique minimal query (up to homomorphic equivalence) [8].

Example - Movies Start with our query and constraint from the introduction:

$$\begin{array}{lll} Q_1 & := & \texttt{for}\ (m_1\ \texttt{in}\ \mathit{Movies})\ (m_2\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \\ & & \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \end{array}$$

$$\begin{array}{lll} C & := & \texttt{forall}\ (x\ \texttt{in}\ \mathit{Movies})\ (y\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{where}\ x.\mathsf{title} = y.\mathsf{title} \\ & & \texttt{exists} \\ & & \texttt{where}\ x.\mathsf{director} = y.\mathsf{director} \end{array}$$

The universal plan, i.e., $\mathit{chase}(C, Q_1)$, is:

$$\begin{array}{lll} U & := & \texttt{for}\ (m_1\ \texttt{in}\ \mathit{Movies})\ (m_2\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \wedge \\ & & \qquad m_1.\mathsf{director} = m_2.\mathsf{director} \\ & & \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor}) \end{array}$$

We proceed with tableau minimization by searching for sub-queries of U. Removing the generator $(m_1\ \texttt{in}\ \mathit{Movies})$ and replacing m_1 with m_2 in the body of Q gives a smaller query:

$$\begin{array}{lll} Q_2 & := & \texttt{for}\ (m_2\ \texttt{in}\ \mathit{Movies}) \\ & & \texttt{return}\ (m_2.\mathsf{director}, m_2.\mathsf{actor}) \end{array}$$

To justify this, we need to check that $C \vdash Q_1 \cong Q_2$, which we can reduce to checking $U = \mathit{chase}(C, Q_1) \leftrightarrow \mathit{chase}(C, Q_2)$. We find that $\mathit{chase}(C, Q_2) \cong Q_2$, so we will actually check that $U \leftrightarrow Q_2$. The identity substitution is a homomorphism $Q_2 \to U$: the important part to notice is the `return` clause, wherein $(m_2.\mathsf{director}$, $m_2.\mathsf{actor})$ is equal to $(m_1.\mathsf{director}$, $m_2.\mathsf{actor})$ precisely because of the equality $m_1.\mathsf{director} = m_2.\mathsf{director}$, which appears in U but not in Q_1. There is also a homomorphism $U \to Q_2$, namely, $m_2 \mapsto m, m_1 \mapsto m$. We thus conclude that $C \vdash U \cong Q_2 \cong Q_1$.

Example - Indexing As we remarked in the introduction, a programmer might be able to optimize our *Movies* query directly, without using the chase at all. But sometimes constraints are not available to the programmer, such as when indices are generated dynamically. Consider the following query, which returns the names of all *People* between 16 and 18 years old:

$$\begin{aligned} Q_1 \quad := \quad & \texttt{for}\ (p\ \texttt{in}\ People) \\ & \texttt{where}\ p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18 \\ & \texttt{return}\ p.\mathsf{name} \end{aligned}$$

Technically, this query is not a purely conjunctive query because the `where` clause involves the less-than predicate $<$. However, the machinery of tableaux minimization can still be used.

Depending on the underlying access patterns, or the whims of a database administrator, an RDBMS might transparently index *People* by creating another relation *Children*, such that the following two constraints hold:

$$\begin{aligned} C_1 \quad := \quad & \texttt{forall}\ (p\ \texttt{in}\ People) \\ & \texttt{where}\ p.\mathsf{age} < 21 \\ & \texttt{exists}\ (c\ \texttt{in}\ Children) \\ & \texttt{where}\ p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \end{aligned}$$

$$\begin{aligned} C_2 \quad := \quad & \texttt{forall}\ (c\ \texttt{in}\ Children) \\ & \texttt{where} \\ & \texttt{exists}\ (p\ \texttt{in}\ People) \\ & \texttt{where}\ p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \end{aligned}$$

In order to use this new index, queries written against *People* must be rewritten to use *Children*. Tableaux minimization provides an automated mechanism to do so.

Let $C = \{C_1, C_2\}$. First, we find the universal plan $U = chase(C, Q_1)$. We begin by chase stepping Q with C_1. The identity substitution is a homomorphism $front(C_1) \to Q_1$, because $p.\mathsf{age} < 21$ is entailed by $p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18$; thus we chase step to:

$$\begin{aligned} U \quad := \quad & \texttt{for}\ (p\ \texttt{in}\ Person)\ (c\ \texttt{in}\ Children) \\ & \texttt{where}\ p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18 \wedge \\ & \qquad p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \\ & \texttt{return}\ p.\mathsf{name} \end{aligned}$$

and we find that $U \cong step(C_1, U)$, so no further chase steps using C_1 are possible. Now we chase step U using C_2, and we find that $U \cong step(C_2, U)$, so no further chase steps with C_2 are possible. Hence we have computed the universal plan $U = chase(C, Q_1)$.

Next, we minimize the universal plan by removing the *Person* generator (note that to do so we must replace each occurrence of p with some other well-typed variable, in this case c):

$$\begin{aligned} Q_2 \quad := \quad & \texttt{for}\ (c\ \texttt{in}\ Children) \\ & \texttt{where}\ c.\mathsf{age} > 16 \wedge c.\mathsf{age} < 18 \\ & \texttt{return}\ c.\mathsf{name} \end{aligned}$$

We now "back-chase" Q_2 with C. We can take no chase steps with C_1, because there is no substitution h that makes $(h(p)\ \texttt{in}\ Person)$ equal to $(c\ \texttt{in}\ Children)$. We can chase step with C_2 using the identity substitution to obtain:

$$\begin{aligned} Q_2' \quad := \quad & \texttt{for}\ (c\ \texttt{in}\ Children)\ (p\ \texttt{in}\ Person) \\ & \texttt{where}\ c.\mathsf{age} > 16 \wedge c.\mathsf{age} < 18 \wedge \\ & \qquad p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \\ & \texttt{return}\ c.\mathsf{name} \end{aligned}$$

At this point, no further steps with C_1 or C_2 are possible. Hence we have computed $Q_2' = chase(C, Q_2)$. Recall that our goal is to check that $C \vdash Q_1 \cong Q_2$, which we do by checking $U = chase(C, Q_1) \leftrightarrow chase(C, Q_2) = Q_2'$; i.e., by checking $U \leftrightarrow Q_2'$. It's easy to prove that U and Q_2' are homomorphically equivalent under the substitution $p \mapsto c, c \mapsto p$, which concludes the optimization.

7. Coq Development - Overview

In the rest of this paper we demonstrate how to shallowly embed relational conjunctive queries into Coq and how to use dependent types and tactics to implement tableaux minimization as described in the first half of this paper. We will continue to use the running movies example from the first half of the paper. That query is expressed in Coq as:

```
Definition Movie : Type := (string × string × string).
Definition Movies : set Movie := . . .

Definition title x := fst x. (* x.title *)
Definition director x := fst (snd x). (* x.director *)
Definition actor x := snd (snd x). (* x.actor *)

Definition q : set (string × string) :=
  m1 ← Movies ; m2 ← Movies ;
  guard (m1.title = m2.title) ;
  return (m1.director, m2.actor).
```

Here, we define a relation `Movies` where each tuple has type `Movie`. We also declare simple functions to access individual fields of the `Movie` type. For consistency with the first half of the paper, we will continue to use dot notation. The next definition defines the query, named `q`. Our Coq query syntax is inspired by Haskell's syntax for monadic computations, and we use Coq's extensible parsing mechanism to define (optional) custom syntax for parsing query expressions. Intuitively, ← means `for`, `guard` means `where`, and `return` means `return`. To the right of the colon is the query's type, `set (string` × `string)`, which represents a set that contains pairs of strings.

Given the above query and a representation of the functional dependency from the introduction (`title_director_ed`) which we will describe shortly, we can ask Coq to automatically construct the minimized query with the following 'proof script':

```
Definition optimized_query:
{q_opt : M (string × string) | title_director_ed → q_opt ≅ q}.
optimize solver.
Defined.
```

The type of `optimized_query` says that `optimized_query` is a pair of a query, $\mathtt{q}_{opt}$, and a proof that $\mathtt{q}_{opt}$ is equivalent to `q` on instances satisfying `title_director_ed`. The actual Coq term corresponding to `optimized_query` is constructed by the `optimize` tactic. We can see the result of the optimization by asking Coq to print the first component of the pair:

```
Eval compute in (proj1_sig optimized_query).
(* = x ← Movies ; return (x.director, x.actor)
 *   : set (string × string)   *)
```

7.1 Queries and Constraints in Coq

We begin by defining a type of sets. Coq's rich type system gives us many possible choices, including sets as lists, `set x := list x`, and sets as predicates, `set x := x → Prop`. In fact, we need not use sets at all: any commutative, idempotent monad with zero defines a type of collections for which the chase is sound [19]. To capture this generality, we use Coq's type class mechanism to

```
Class DataModel (M : Type → Type) : Type :=
{ Mret  : ∀ {T}, T → M T
; Mbind : ∀ {T U}, M T → (T → M U) → M U
; Mzero : ∀ {T}, M T
; Mimpl : ∀ {T}, M T → M T → Prop
; Mprod : ∀ {T U}, M T → M U → M (T * U) :=
     fun _ _ m1 m2 ⇒ Mbind m1 (fun x ⇒ Mbind m2 (fun y ⇒
      Mret (x,y)))
; Mguard : ∀ {T}, bool → M T → M T :=
     fun _ P m ⇒ if p then m else Mzero
 (* plus many axioms *)
}.

(* Queries *)
Definition query {S T: Type}
  (P : M S) (C : S → bool) (E : S → T) : M T :=
  Mbind P (fun x ⇒ Mguard (C x) (Mret (E x))).

(* Embedded Dependencies *)
Definition embedded_dependency {S S': Type}
  (F : M S) (Gf : S → bool) (B : M S') (Gb : S → S' → bool)
:= Meq (query F Gf (fun x ⇒ x))
       (query (Mprod F B)
              (fun ab ⇒ Gf (fst ab) && Gb (fst ab) (snd ab))
              (fun x ⇒ fst x)).
```

Figure 1. The `DataModel` and the definitions of `for-where-return` queries and embedded dependencies.

express “chaseable monads” (see Figure 1) which we will refer to using the type constructor `M`. The operations, but not the required axioms, of our monadic interface are shown in Figure 1. Intuitively, in a set monad these operations are:

- `Mret v` is the singleton set containing only `v`.
- `Mbind m k` is the function that unions all sets `k x` for every `x` in the set `m`. We will write `x ← m ; k` for `Mbind m (fun x ⇒ k)` where `x` occurs free in `k`.
- `Mzero` is the empty set.
- `Mimpl` m_1 m_2 states that m_1 is a subset of m_2. We extend subset to equivalence (written `Meq` m_1 m_2) using mutual containment.

Using these as primitives, we define two additional operations.

- `Mprod` is the Cartesian product of two sets. (In an arbitrary monad we require a *strength* [3].)
- `Mguard P m` is defined as `Mzero`, if `P` is false and `m` otherwise.

On top of the data model we define `for-where-return` queries (Figure 1). In the definition, `P` represents the `for` clause, `C` represents the `where` clause, and `E` represents the `return` clause. Concretely, the movies query from the introduction is represented as:

```
query (Mprod Movies Movies)
      (fun ab ⇒ (fst ab).title ?= (snd ab).title)
      (fun ab ⇒ ((fst ab).director, (snd ab).actor))
```

Note that the S and T arguments are omitted since they can be inferred from the other arguments (e.g., S is determined by P). P constructs the product of the $Movies$ relation with itself. The `where` clause is a function that accepts a pair where the first element (`fst ab`) comes from the first copy of $Movies$ and the second (`snd ab`) comes from the second copy of $Movies$; the `where` clause checks equality using Coq's boolean-valued equality operator which we denote with ?=. The `return` clause returns a pair of strings constructed from the pair of $Movies$.

We define embedded dependencies (Figure 1) using the $front = back$ definition from Section 3. Here, `F` is the `for` clause of the front, `Gf` is the `where` clause of the front, `B` is the `exists` clause of the back, and `Gb` is the `where` clause of the back. In this representation, the ED for the movies example is the following:

```
Definition title_implies_director : Prop :=
  embedded_dependency
    (Mprod Movies Movies)
    (fun ab ⇒ (fst ab).title ?= (snd ab).title)
    (Mret tt)
    (fun ab _ ⇒ (fst ab).director ?= (snd ab).director).
```

Here, the empty `exists` clause is represented by a singleton set carrying the unit value `tt`.

Query Normalization In the first half of the paper, all of our queries were normalized into a single `for` clause, a single `where` clause, and a single `return` clause. But in a language-integrated query system, we can relax this requirement. For example, we could introduce an additional guard condition after binding `m1`.

```
Definition q_LOR : set (string × string) :=
  m1 ← Movies ;
  guard (m1.title ?= ‘‘Lord of the Rings’’) ;
  m2 ← Movies ;
  guard (m1.title ?= m2.title ) ;
  return (m1.director, m2.actor).
```

As is well-known [13], monadic computations such as relational conjunctive queries can always be normalized into the flat form used in the first half of this paper, and our optimizer performs this normalization (in a fully verified way) during optimization.

7.2 Background: Tactic-based Programming

The core of the Coq Proof Assistant is a pure and dependently-typed functional programming language, called Gallina. In addition to Gallina, Coq also includes a Turing-complete, untyped “tactic” language, called $\mathcal{L}_{\text{tac}}$, which can be used to construct Gallina terms in a semi-imperative style. $\mathcal{L}_{\text{tac}}$ is a meta-language for Gallina in much the same way that macros are a meta-language for C++. $\mathcal{L}_{\text{tac}}$ tactics generate Gallina terms that are checked by the Coq kernel in the same way that hand-written terms are checked.

Remark. Our query optimizer is fully automated: to use it, a user simply invokes a tactic called `optimize`. By the completeness theorem for the chase [19], `optimize` will find the minimal equivalent query or diverge when none exists. The details of our Coq implementation that we now discuss describe the design decisions we made when building our optimizer are not necessary for clients of our optimizer to understand. However, many of the techniques are general-purpose and can be re-used in other developments which use tactics for similar purposes.

The $\mathcal{L}_{\text{tac}}$ Programming Model In this paper, we use $\mathcal{L}_{\text{tac}}$ to optimize queries in a similar way to Fiat's use of $\mathcal{L}_{\text{tac}}$ to optimize programs [7]. This use of $\mathcal{L}_{\text{tac}}$ is somewhat unorthodox; the standard use of $\mathcal{L}_{\text{tac}}$ is to prove theorems via “proof scripts” that express how proofs should be built. Indeed, the operational semantics of $\mathcal{L}_{\text{tac}}$ are tailored to fit this standard use of $\mathcal{L}_{\text{tac}}$, rather than to fit our use of $\mathcal{L}_{\text{tac}}$ as a query optimizer. For example, every $\mathcal{L}_{\text{tac}}$ tactic runs against a particular “proof obligation” (a.k.a goal), such as `x + y = y + x`. In general, these goals are fully determined and it is simply the tactic's job to find a suitable proof. In order to use $\mathcal{L}_{\text{tac}}$ to optimize queries we will run our tactics on partially specified goals which specify a constraint on a “unification variable” that the tactic will seek to fill in while constructing a proof. The values of these unification variables will be the optimized queries.

We will demonstrate our use of $\mathcal{L}_{\text{tac}}$ on a simple goal, where we want to instantiate the unification variable ?n with an optimized version of the query on the right:

```
Meq ?n (x ← Movie ; y ← Movie ; ret x)
```

We think of the form of this goal as a "calling convention" of sorts for the optimization. Here, ?n is the unification variable that we are trying to fill in and the constraint is that the query we pick must be equivalent to the query 'x ← Movie ; y ← Movie ; ret x'. We could simply pick ?n by reflexivity to be the full query, but that would require the database execution engine to do an unnecessary join. Instead, we will optimize the query on the right by appealing to various transformations that are provably correct.

Optimization via Rewriting One simple way to perform optimization is through rewriting. For example, the following lemma expresses that queries are idempotent (note that the second generator is not used in the rest of the query):

```
Lemma Mbind_dedup : ∀ {T U} (m : M T) (k : T → M U),
  Meq (Mbind m k) (Mbind m (fun x ⇒ Mbind m (fun _ ⇒ k x))).
Proof. . . . Qed.
```

Because Meq is an equivalence relation, we can use $\mathcal{L}_{\text{tac}}$'s setoid rewriter to rewrite the goal using Mbind_dedup. Running setoid_rewrite ← Mbind_dedup on the goal above results in the new goal:

```
Meq ?n (x ← Movie ; Mret x)
```

Note that this does *not* solve the unification variable ?n or the goal; only the right-hand side of the Meq is changed yielding a new goal to operate on. Because no more optimization is possible, we solve the goal by appealing to the reflexivity of Meq which implicitly instantiates ?n with 'x ← Movie ; Mret x'. Note that the optimized query no longer requires the extraneous join.

Optimization via Applying Lemmas Rewriting is a useful reasoning technique because the rewriting machinery is often able to automatically construct the proofs necessary to justify the manipulation deep within the term. However, because the above reasoning is occurring at the top level of the goal, we could also use $\mathcal{L}_{\text{tac}}$'s more primitive eapply tactic with Mbind_dedup to yield a proof in a single step. While both rewriting and applying lemmas are useful for manipulation, we find that most interesting optimization is better phrased using application because it is more predicable. In large goals, rewriting could occur anywhere within the term, whereas lemma application will only apply at the top level.

In building up larger automation procedures using eapply, we use lemmas expressed so that their conclusion matches the current goal and their premises express new constraints that we will pass to other tactics. Consider the following (slightly contrived) example:

```
Meq ?n (Mprod a b)
```

If we wish to optimize a and b independently then we can eapply the following lemma:

```
Lemma opt_plus : ∀ {T U} (m1 m1' : M T) (m2 m2' : M U),
  Meq m1' m1 →
  Meq m2' m2 →
  Meq (Mprod m1' m2') (Mprod m1 m2).
```

When we eapply this lemma to the above goal, we expect m1 and m2 to be completely determined while m1' and m2' to be new unification variables. Concretely, when we run eapply opt_plus on the above goal, we get the following two sub-goals:

```
Meq ?m1' m1        Meq ?m2' m2
```

while at the same time instantiating ?n with Mprod ?m1' ?m2'. As our automation continues to fill in ?m1' and ?m2' (for example, during further optimization), ?n will be automatically updated to reflect these changes.

7.3 Implementing the Chase as a Tactic

Tableaux minimization makes heavy use of the chase, which we have implemented as an $\mathcal{L}_{\text{tac}}$ tactic. Implementing the chase as a tactic (as opposed to a Gallina function) has two critical advantages. First, *Coq tactics can invoke other Coq tactics.* As we saw in the indexing example in the first half of the paper, running the chase can involve reasoning over predicates such as $<$ that appear in where clauses. By implementing the chase as a tactic, we can appeal to Coq's omega tactic for reasoning about $<$, for example. Moreover, the particular predicates and associated tactics need not be determined in advance; users are free to write arbitrary Gallina code in where clauses, and to build custom tactics for reasoning about this code. The second reason is that we are not obligated to justify termination of $\mathcal{L}_{\text{tac}}$ scripts. The chase may never terminate, but all Gallina functions must terminate, so any Gallina (but not $\mathcal{L}_{\text{tac}}$) implementation of the chase must be explicitly bounded by some number of chase steps. (If desired, our $\mathcal{L}_{\text{tac}}$ version of the chase can also be given an explicit bound).

Our chase tactic applies to goals that are contingent on EDs. These EDs are expressed using the following goal structure:

```
title_director →
Meq ?q
    (m1 ← Movies ; m2 ← Movies ;
     guard (m1.title = m2.title) ;
     return (m1.director, m2.actor))
```

Recall from the first half of the paper that running the chase required several steps: choosing an ED, finding a candidate substitution, checking that a candidate is in fact a homomorphism, taking a chase step, and then checking for convergence to a fixed point. In the rest of this section we describe how to implement each of these in $\mathcal{L}_{\text{tac}}$. In the process we distill reusable patterns for building optimization procedures in $\mathcal{L}_{\text{tac}}$.

7.3.1 Finding an Embedded Dependency

In the movies example, there is only one embedded dependency to chase with, but in more complex examples such as the indexing example there are multiple EDs. To find an appropriate ED, we can exhaustively consider all of the user-supplied EDs. Demonstrating this exhaustive enumeration serves as a useful primer for the more complicated next step of finding a homomorphism. Indeed, this section describes a *general-purpose design pattern* for doing exhaustive enumeration using $\mathcal{L}_{\text{tac}}$.

When there are multiple candidate EDs (e.g., A, B, and C) that we wish to use to optimize the query q, the goal posed to the optimizer tactic has the following form:

```
A ∧ B ∧ C → Meq ?q q
```

Two lemmas allow us to exhaustively consider these EDs.

```
Lemma ed_pick_left : ∀ {A B C}, (A → C) → (A ∧ B → C).
Proof. tauto. Qed.
Lemma ed_pick_right : ∀ {A B C}, (B → C) → (A ∧ B → C).
Proof. tauto. Qed.
```

Both of these lemmas are trivial tautologies that can be proven by the tauto tactic; however, we have found that separating the task of crafting each lemma from its proof dramatically improves our ability to evolve our optimizer. ed_pick_left focuses on the left-hand-side of a conjunction while ed_pick_right focuses on the right-hand-side. When only a single ED remains, we can process it, in this case chasing with it, which we will discuss in more detail in the next section.

We use these lemmas in the following backtracking search tactic which uses $\mathcal{L}_{\text{tac}}$'s + combinator[1] in the following recursive tactic.

```
Ltac ed_search :=
  lazymatch goal with
  | ⊢ _ ∧ _ → _ ⇒
    (  simple eapply ed_pick_left
     + simple eapply ed_pick_right) ; ed_search
  | ⊢ _ → _ ⇒ idtac
  end.
```

Here, '`lazymatch goal with`' performs syntactic matching of the goal against the candidate patterns selecting the first that matches. The first branch chooses either the left- or right-side using the above lemmas and then continues the search by recursively calling `ed_search`. The second branch is the default case which finishes the search when the premise does not contain a conjunction using `idtac` which is a no-op. It is important to note that the semantics of + causes deep backtracking. For example, `(a + b) ; c` is equivalent to `(a; c) + (b; c)`. This behavior allows us to chain `ed_search` with the chase (described in the next section) and backtrack to consider a new ED if the chase step fails to make progress with the first ED.

7.3.2 Running the Chase Step

Once we have isolated a single ED to chase, we need to apply a theorem witnessing the soundness of the chase. This soundness theorem is phrased to match the current goal.

```
1  Theorem chase_sound {S S' T U}
2    (P : M S) (C : S → bool) (E : S → T)
3    (F : M S') (Gf : S' → bool) (B : M U) (Gb : S' → U → bool)
4  : ∀ (h : S → S'),
5      Mimpl (Mmap h P) F →
6      (∀ x, C x = true → Gf (h x) = true) →
7      embedded_dependency F Gf B Gb →
8      Meq (query P C E)
9          (query (Mprod P B)
10                (fun ab : S × U ⇒ C (fst ab) &&
11                                Gb (h (fst ab)) (snd ab))
12                (fun ab ⇒ E (fst ab))).
```

In this theorem, lines 8-12 will unify with the goal. Line 8 and the first argument to `Meq` extract the values of the goal and the second-argument to `Meq` instantiates the unification variable with the optimized query. Line 4 represents the substitution (h). Line 5 expresses the requirement that h maps variables in the query (P) to the variables in the front of the ED (F). Line 6 represents the requirement on the `where` clause, namely that the `where` clause of the query (C) implies the `where` clause of the front of the ED (Gf). In the next two sections we describe the representation of these pieces in more detail and explain how we compute this homomorphism.

Finding a Homomorphism In the definition of a chase step in Section 5 a homomorphism is a map from variables bound in the front of the embedded dependency to the variables bound in the `for` clause of the query. Since our queries are Gallina terms, explicitly referencing binders by name can be quite difficult, and is not very extensible (i.e., not invariant under α-renaming). Instead, we encode our mapping of binders as a function between the types being bound. In this example, the type of the `for` clause of the query is `Movie` × `Movie` and the type of the `forall` clause is also `Movie` × `Movie`, so we are looking for a function `h : Movie` × `Movie` → `Movie` × `Movie`. Looking at the query and the ED, there are four choices:

```
h x = (fst x, fst x)    h x = (snd x, fst x)
h x = (fst x, snd x)    h x = (snd x, snd x)
```

We are going to construct these functions incrementally using theorems that represent individual steps of reasoning. The search is much the same as the search for isolating a particular ED, but with two main differences. First, we must now find a binder for each binder in the front of the ED, i.e. we must perform a new search for each binder. Second, while doing this, *we must explicitly construct the substitution* `h` so that we can use it when checking the remainder of the homomorphism (line 6 in `chase_sound`). When searching for the homomorphism, the goal will have the following form, where `P` is the `for` clause of the query and `F` is the `forall` clause in the ED:

```
Mimpl (Mmap ?h P) F
```

In the above `Mmap` expresses the application of the substitution `?h`, i.e. `Mmap f m = x ← m ; Mret (f x)`. Note that here we are proving an inclusion (`Mimpl`) rather than an equality. This is essential because fields not used to construct F from P could be empty which would make the left-hand side empty while the right-hand side would be non-empty.

When solving this goal, the first task is to break `F` down into atomic units that correspond to the binders. The `pick_split` lemma applies when `F` is formed from a `Mprod`:

```
Lemma pick_split
: ∀ {T U U' : Type} (m : M T) (u : M U) (u' : M U') f g,
  Mimpl (Mmap f m) u →
  Mimpl (Mmap g m) u' →
  Mimpl (Mmap (fun x ⇒ (f x, g x)) m) (Mprod u u').
Proof. . . . Qed.
```

This lemma states that we can find a morphism from `m` to `Mprod u u'` if we can find a morphism from `m` to `u` and from `m` to `u'`. Note that in addition to breaking down the morphism by decomposing it into `f` and `g`, the left-hand side of the implication in the conclusion also shows how to use `f` and `g` to construct the final homomorphism.

Repeatedly applying `pick_split` will eventually break the `forall` clause down into atomic elements that we can match up with the query. This matching is essentially the same as the ED search procedure except that, as above, we must record the way to reconstruct the `h` function. The following three lemmas express (and justify) these manipulations.

```
Lemma pick_left
: ∀ {T' U' V} (f' : U' → V) (x : M V) (y : M T') (k' : M U'),
  Mimpl (Mmap f' k') x →
  Mimpl (Mmap (fun x ⇒ f' (fst x)) (Mprod k' y)) x.
Proof. . . . Qed.
Lemma pick_right
: ∀ {T' U' V} (f' : U' → V) (x : M V) (y : M T') (k' : M U'),
  Mimpl (Mmap f' k') x →
  Mimpl (Mmap (fun x ⇒ f' (snd x)) (Mprod y k')) x.
Proof. . . . Qed.
Lemma pick_here
: ∀ {T} (x : M T), Mimpl (Mmap (fun x ⇒ x) x) x.
Proof. . . . Qed.
```

`pick_left` decides to use only the left-hand side of the `Mprod k' y` to determine `x`, `pick_right` is analogous for the right-hand side. Finally, `pick_here` applies when the value being searched for is exactly the value being bound in which case it can pick the value directly.

[1] The + combinator is new in Coq 8.5. Before Coq 8.5, performing this search modularly required writing tactics in continuation-passing style.

The search completes by solving the goal and, by side-effect, instantiating ?h with a function representing the substitution.

Proving the Side-Conditions With a candidate substitution in hand, the next step is to discharge the side condition which ensures that the `where` clause of the embedded dependency implies the `where` clause of the query. In our movies example, this amounts to the following:

```
∀ x : Movie × Movie, (fst x).title ?= (snd x).title = true
    → (fst (h x)). title ?= (snd (h x)).title
```

Once we get to this step, `h` is exactly one of the substitutions constructed by the previous step. When we plug in a particular substitution, i.e. '`h x = (fst x, snd x)`' (recall that we are enumerating all of the potential homomorphisms), and simplify, we are left to solve the following goal:

```
∀ x : Movie × Movie, (fst x).title ?= (snd x).title = true
    → (fst x). title ?= (snd x).title = true
```

While this goal is true simply by equational reasoning, in general these side conditions can require more complex reasoning. For example, in the indexing example from Section 6 we must prove the following implication which relies on arithmetic reasoning:

```
∀ p, p.age > 16 && p.age < 18 = true → p.age < 21 = true
```

We can use Coq's `omega` tactic to discharge arithmetic goals such as the one above. In order to facilitate this sort of domain-specific reasoning, we parameterize the chase tactic by a tactic to discharge side-conditions.

Ensuring Progress After the side-condition is checked, the final step is to ensure that this chase step makes progress by adding *new* structure to the query. Semantically, we express this by ensuring that the new query is not homomorphically equivalent to the old one. Checking homomorphic equivalence between queries is essentially the same as what we have done up to this point except that we must also show that the morphism preserves the `return` clause of the query under the assumptions in the `where` clause. The $\mathcal{L}_{\text{tac}}$ to check this condition is straightforward given the machinery that we developed up to this point. The entire $\mathcal{L}_{\text{tac}}$ is (essentially) the following:

```
Ltac prove_query_morphism solver :=
  eapply check_query_morphism_apply ;
    [ find_bind_morphism
    | simpl ; solve [ solver ]
    | simpl ; solve [ solver ] ]) .

Ltac prove_query_homomorphic_equal solver :=
  split; prove_query_morphism solver.
```

Here chaining the `eapply` with `; [ a | b | c ]` allows us to specify different tactics to run on each of the individual goals produced by `eapply check_query_morphism_apply`. Note again that these tactics are parameterized by the underlying solver (`solver`) that they will use to discharge the side-conditions.

7.3.3 Computing the Fixed-point

With the ability to iterate over EDs (Section 7.3.1) and to compute a chase step (Section 7.3.2), it is simple to implement the entire chase. The tactic is the following:

```
Ltac chase solver :=
  repeat first
    [ eapply transitive_refine_conditional ;
      [ solve [ ed_search ; chase_step solver ]
      | ]
    | reflexivity ].
```

where `transitive_refine_conditional` expresses transitivity under implication:

```
Lemma transitive_refine_conditional
: ∀ {T} (a b c : M T) (P : Prop),
    (P → Meq b c) → (P → Meq a b) → (P → Meq a c).
Proof. . . . Qed.
```

In the `chase` tactic, `repeat` computes the transitive closure of the chase step by running the rest of the tactic until the goal is solved or the tactic fails. `first [ a | b ]` runs `a` and, if `a` fails, runs `b`. The first branch uses the transitivity lemma above to break the goal into two sub-goals. For the first sub-goal, we run a single step of the chase by first finding an embedded dependency (`ed_search`) and then chasing it using `solver` to solve the side conditions (`chase_step solver`). The `solve` ensures that the tactic completely solves the goal which guarantees that we completed the chase step. If the goal is solved, the second goal is interpreted by the rest of the tactic due to the `repeat`. If the chase step fails, then the second branch of the `first` runs solving the goal using `reflexivity` which picks the input query to be the output query.

```
Lemma minimize_drop
: ∀ {T T' V : Type} (qb : M T) (qb' : M T') qg (qr : _ → V) f
     (qb'' : M T') qg'',
   Find f
→ Meq (query (Mprod qb qb') qg qr)
      (query qb' (fun y ⇒ qg (f y,y)) (fun y ⇒ qr (f y,y)))
→ Meq (query qb' (fun y ⇒ qg (f y,y)) (fun y ⇒ qr (f y,y)))
      (query qb'' (fun y ⇒ qg'' (f y,y)) (fun y ⇒ qr (f y,y)))
→ Meq (query (Mprod qb qb') qg qr)
      (query (Mmap (fun y ⇒ (f y, y)) qb'') qg'' qr).
Proof. . . . Qed.

Lemma minimize_keep
: ∀ {T T' V : Type} (qb : M T) (qb' : M T') qg (qr : _ → V) (
     qb'' : M T') qg'',
  (∀ x : T,
   Meq (query qb' (fun y ⇒ qg (x,y)) (fun y ⇒ qr (x,y)))
      (query qb'' (fun y ⇒ qg'' (x,y)) (fun y ⇒ qr (x,y)))) →
  Meq (query (Mprod qb qb') qg qr)
     (query (Mprod qb qb'') qg'' qr).
Proof. . . . Qed.
```

Figure 2. Minimization lemmas.

7.4 Query Minimization

The final step in optimization is to remove redundant generators (binds) from the query. By now, we have already discussed most of the techniques. The high-level approach is to use incremental lemmas to iterate through the binders and attempt to drop each one by expressing a side-condition that expresses that the information in that binder can be reconstructed from the other binders.

There are two core lemmas that express minimization transformations (shown in Figure 2). The first, `minimize_drop`, states that we can drop the first (left) binder if we can find a way to compute it from the right binder, i.e. a morphism from `Mmap f qb'` to `qb`. The first premise of this theorem, `Find f`, is a dummy premise; it is trivially true. We include it in order to simplify constructing `f` using tactics. For example, we can easily write lemmas analogous to `pick_left`, `pick_right`, and `pick_here` looking only at the type of `f`. The second premise, ensures that this choice of `f` respects the equivalence of the query. To solve it, we "back chase" the left-hand side of the equivalence and try to determine if it is homomorphically equivalent to the right-hand side. The second lemma, `minimize_keep`, is the fallback case. If the tactic can not find a way to re-construct the removed binder from the other binders of the query, then it cannot remove that binder. However, it can (and should) still optimize the rest of the query. To represent the rest of the query, we universally quantify over the values that could come from the relation and recursively optimize the rest of the query.

Post-processing The primary purpose of the `optimize` tactic is to minimize the number of binds in a query, but we have also added some query simplification steps to be performed after minimization. In the movies example, the result of minimization is:

```
query (Mmap (fun x ⇒ (x,x)) Movie)
        (fun xy ⇒ (fst xy).title = (snd xy).title)
        (fun xy ⇒ ((fst xy).director, (snd xy).actor))
```

After simplification, the duplication of the bound variable is removed and the `where` clause is eliminated since it is testing the equality of `x.title` with itself. The final query is the following:

```
query Movie (fun _ ⇒ true) (fun x ⇒ (x.director, x.actor))
```

8. Discussion of Tactic-based Optimization

Tactic-based development has several trade-offs compared to a more traditional approach that would implement a query optimizer as a Gallina function (e.g., [4]).

Benefits The primary benefit of implementing the optimizer as a tactic is the ability to extend the optimizer with a minimal amount of work. Much of this benefit is due to the flexibility of working indirectly on Gallina's underlying terms. For example, we can extend the chase algorithm with support for nested relations by proving new lemmas that show how to locally manipulate terms. A non-tactic implementation would need to adjust its concrete term representations to support the more sophisticated structure of nested queries and then update all of its algorithms (and proofs) to work on the new nested representation. Similarly, we explicitly support arbitrary Coq computation within `where` clauses. While the examples in this paper use only equality ($=$) and less-than ($<$) in `where` clauses, there is nothing preventing us from reasoning about more complex operations (e.g., case-insensitive string comparison).

Another benefit of the tactic-based approach is that we are able to re-use a considerable amount of Coq's underlying infrastructure. Features such as higher-order unification, existing automation libraries, and $\mathcal{L}_{\text{tac}}$'s backtracking search mechanism are all useful when building a query optimizer. We have found, in particular, that the new backtracking proof search facilities, namely + and dependent goals, introduced in Coq 8.5, are extremely useful when building tactics such as these. For example, without this backtracking feature, tactics that wish to backtrack must be written in continuation passing style, and even then it can be quite difficult to maintain all the necessary information.

Drawbacks There are also drawbacks to using tactics to implement a procedure as sophisticated as semantic optimization. First, tactics are completely untyped which makes it cumbersome to track down errors, which are often due to simple typos. While simple types would help track some information, throughout the course of development we found that one of the most cumbersome tasks was keeping track of simple properties about goals; for example, whether the unification variable to be constructed was on the left or the right of an `Meq`. Similarly, many lemmas had to be duplicated to handle extra bits of context; for example, we had to write separate lemmas for chaining together refinements in the presence and absence of embedded dependencies. Several authors have proposed more richly typed tactic-based programming languages, notably Mtac [23] and VeriML [20], and while neither of these are as mature or rich as $\mathcal{L}_{\text{tac}}$, it would be interesting to explore whether their features would be useful in our development process.

	Time (s)	
Phase	**Movie**	**Index**
Normalize	0.64	0.48
Chase	0.89	3.9
Minimize	1.14	34.14
Simplify	0.09	0.23
Total	3.50	38.8

Figure 3. Performance of the optimizer.

Another problem inherent to $\mathcal{L}_{\text{tac}}$ is speed. Figure 3 shows the time it takes to optimize the queries presented in this paper using a Intel Core i5-4460 CPU at 3.20GHz on Coq 8.5 beta 2. The `normalize` task (not discussed) is the time it takes to normalize Coq terms, using the monad laws, into the flat form `query P C R`. The `chase` and `minimize` phase are the phases described in Sections 7.3 and 7.4. The final `simplify` phase performs the rudimentary non-semantic optimization, discussed briefly at the end of Section 7.4. The Total row is the total of all of the phases run from beginning to end with no other timing or intermediate results. Overall, the optimizer is somewhat slow, especially on the index example. In this case, a non-negligible fraction of the overall time in the index example can be attributed to solving numeric side conditions using the `omega` tactic.

There are a few caveats to keep in mind when interpreting these performance results:

- Finding a homomorphism between two (conjunctive) tableaux is an NP-hard problem [8]. Specialized heuristic algorithms exist to solve this problem quickly, but our implementation uses a naïve brute-force search.
- Coq's $\mathcal{L}_{\text{tac}}$ implementation is both interpreted and single-threaded.
- Much optimization is done offline where performance is not essential.

One way to make our query optimizer faster is to implement pieces of it directly as Coq programs (similar to [4]). Indeed, our initial implementation was a Gallina function rather than a tactic and it ran nearly instantaneously on the movies query. However, implementing the optimizer entirely as a Gallina program suffers from the drawbacks discussed above. In particular, scaling it to the index example would have required implementing (and proving sound) a Gallina function to reason about numbers and $<$. While in theory such a procedure would only need to be implemented once, composing Gallina functions such as these has been a notoriously difficult problem which has only recently been addressed [17]. Finding the exact balance between the two approaches is a promising direction for future work.

Coq as a DBPL Although our focus in this paper is on the semantic optimization of queries shallowly embedded in Coq, it is worth reflecting more broadly on the use of Coq as a host language for embedded queries. The biggest drawback of Coq in this context is its purity. Establishing a connection to a database is invariably an effectful operation, and in Coq such operations must be defined as axioms and segregated behind a monadic interface [18]. Coq can extract such effectful code into OCaml or Haskell where it can be executed, but the Coq kernel itself cannot execute effectful code.

On the positive side, Coq inherits all of the usual advantages of a strongly-typed functional programming language for hosting a query language [13]. Moreover, Coq's dependent types can be used to express computations that cannot be (safely) expressed in languages with weaker type systems. For example, it is possible to define a Coq function f that can only be called on an input instance I that is known to satisfy some constraint C using the following pseudo-code:

```
Definition f (C: ED) I (pf: holds I C) := ...
```

Indeed, the constraint C need not even be known at compile time.

9. Conclusion

In this paper we have described a verifying semantic query optimizer for Coq based on the chase [8]. Our approach leverages programming with tactics to manipulate raw Coq terms and simultaneously produce an optimized query and a proof that the query has the same meaning as the input query. Implementing the optimizer as a tactic has many advantages, including, first and foremost, essentially limitless extensibility: because Coq tactics can invoke other tactics, users are free to plug-in their own proof automation to be used during the optimization process. For example, one of our examples makes use of Coq's `omega` tactic for reasoning about natural numbers and the less than relation $<$. As far as we are aware, no other work on formalizing the relational model in Coq (e.g., [4]) implements the chase as a tactic.

However, implementing query optimization as a tactic does pose certain challenges, most of which arise because $\mathcal{L}_{\mathrm{tac}}$ was designed with proof scripting in mind, rather than query optimization. To work in this environment we construct terms incrementally using unification variables and express properties about them using goals. These goals form a sort of "calling convention" for tactics and we use lemmas to perform incremental reasoning either through rewriting or direct application. We leverage backtracking search to explore multiple potential optimization paths. In phrasing problems dealing with binders we express syntactic manipulation of binders extensionally as functions that operate on environments. We hope that our solutions to these issues in this paper will allow others to create their own query (and program) optimizers as Coq tactics.

While not currently competitive with more traditional programming languages in terms of speed, $\mathcal{L}_{\mathrm{tac}}$ is continually improving and similar systems feature prominently in the development of large-scale verified software systems such as Idris [6] and Agda [2]. Moreover, recent work [9, 12, 22] has shown how tactic-based programming can be made first-class (i.e., reified into the language) by exploiting dependent types. This next wave of tactic-based languages and libraries are already demonstrating substantial performance improvements and it is likely that tactic-based optimization will soon be a viable design pattern for language-integrated query systems.

Acknowledgement The authors would like to thank Lucian Popa for answering many questions about semantic optimization.

References

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

[2] Agda Development Team. The Agda proof assistant reference manual, version 2.4.2. 2014.

[3] Michael Barr and Charles Wells, editors. *Category theory for computing science, 2nd ed.* 1995.

[4] Vronique Benzaken, velyne Contejean, and Stefania Dumbrava. A Coq Formalization of the Relational Data Model. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 189–208. Springer Berlin Heidelberg, 2014.

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[7] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *POPL'15*, pages 689–700, 2015.

[8] Alin Deutsch, Lucian Popa, and Val Tannen. Query reformulation with constraints. *SIGMOD Rec.*, 35:65–73, March 2006.

[9] Dominique Devriese and Frank Piessens. Typed Syntactic Metaprogramming. ICFP '13, pages 73–86, New York, NY, USA, 2013. ACM.

[10] Tom Ellis. Opaleye. Technical report, http://github.com/tomjaguarpaw/haskell-opaleye.

[11] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of CS, University of Nottingham, November 1996.

[12] Gregory Malecha. *Extensible Proof Engineering in Intensional Type Theory*. PhD thesis, Harvard University, 2014.

[13] Torsten Grust. *Monad Comprehensions. A Versatile Representation for Queries. In The Functional Approach to Data Management, P.M.D. Gray and L. Kerschberg and P.J.H. King and A. Poulovassilis (eds.).* Springer Verlag, 2003.

[14] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, 2005.

[15] Qi heng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. VLDB, 1999.

[16] S. Kazem Lellahi and Val Tannen. A calculus for collections and aggregates. In *CTCS '97*, 1997.

[17] Gregory Malecha, Adam Chlipala, and Thomas Braibant. Compositional Computational Reflection. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 374–389. Springer International Publishing, 2014.

[18] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. POPL '10, pages 237–248, New York, NY, USA, 2010. ACM.

[19] Lucian Popa and Val Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *ICDT*, 1999.

[20] Antonis Stampoulis and Zhong Shao. VeriML: Typed Computation of Logical Terms Inside a Language with Effects. ICFP '10, pages 333–344, New York, NY, USA, 2010. ACM.

[21] Val Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. ICDT '92, pages 140–154, London, UK, 1992. Springer-Verlag.

[22] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 157–173. Springer Berlin Heidelberg, 2013.

[23] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. In *ICFP'13*, ICFP '13, pages 87–100, New York, NY, USA, 2013. ACM.