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

Gregory Malecha, gmalecha@eng.ucsd.edu
Ryan Wisnesky, wisnesky@math.mit.edu

DBPL
October 27, 2015

Outline

- Goal: build a query optimizer in Coq
  - not to prove it correct, but
  - to optimize monad comprehensions
    - toward dependently-typed LINQ!
- I will describe:
  - the basics of conjunctive query optimization
  - how to represent data integrity constraints in Coq
  - how to build a query optimizer as a Coq tactic
- Who cares?
  - Coq users can use our tactic to optimize monad comprehensions in a provably correct way.
  - Our work gives a *design pattern* for optimizing Coq code using tactics.
- Talk goals:
  - Introduce semantic query optimization to functional programmers
  - Introduce dependently-typed programming to database specialists
  - The details of the Coq tactic are too difficult to convey in a talk

Overview

- Part 1:
  - Given a relational conjunctive query Q
  - and a set of constraints C of the form $\forall \vec{x}.\phi(\vec{x}) \rightarrow \exists \vec{y}.\psi(\vec{x}, \vec{y})$
  - we can compute a unique minimal query Q' such that $C \vdash Q \cong Q'$
  - or diverge

- Part 2:
  - Given a commutative, idempotent monad with zero in Coq
  - and a Coq monad comprehension Q
  - and a set of Coq proof objects C
  - our Coq tactic (semi) computes Q' and a proof that $C \vdash Q \cong Q'$

Semantic (constraint-aware) optimization

- Return tuples (d, a) where a acted in a movie directed by d.

$$\texttt{for}\ (m_1\ \texttt{in}\ \mathit{Movies})\ (m_2\ \texttt{in}\ \mathit{Movies})$$
$$\texttt{where}\ m_1.\text{title} = m_2.\text{title}$$
$$\texttt{return}\ (m_1.\text{director}, m_2.\text{actor})$$

- Under functional dependency title → director is equivalent to:

$$\texttt{for}\ (m\ \texttt{in}\ \mathit{Movies})$$
$$\texttt{return}\ (m.\text{director}, m.\text{actor})$$

Embedded Dependencies (EDs)

- Let P and B be conjunctions of equalities (e.g., $x_1 = x_2$) and memberships (e.g, $R(x_1, x_2)$):

$$\texttt{forall}\ \overrightarrow{(x\ \texttt{in}\ X)}$$
$$\texttt{where}\ P(\overrightarrow{x})$$
$$\texttt{exists}\ \overrightarrow{(y\ \texttt{in}\ Y)}$$
$$\texttt{where}\ B(\overrightarrow{x}, \overrightarrow{y})$$

- Functional dependency title → director expressed as:

$$\texttt{forall}\ (x\ \texttt{in}\ \mathit{Movies})\ (y\ \texttt{in}\ \mathit{Movies})$$
$$\texttt{where}\ x.\text{title} = y.\text{title},$$
$$\texttt{exists}$$
$$\texttt{where}\ x.\text{director} = y.\text{director}$$

The front and back of an ED

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

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

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

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

Homomorphisms of queries

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

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

- is a substitution $\vec{v_1} \mapsto \vec{v_2}$ such that
  - $\overrightarrow{(h(v_1) \texttt{ in } V_1)} \subseteq \overrightarrow{(v_2 \texttt{ in } V_2)}$
  - $P_2(\vec{v_2}) \vdash P_1(h(\vec{v_1}))$
  - $P_2 \vdash R_1(h(\vec{v_1})) = R_2(\vec{v_2})$
- $Q_1 \to Q_2$ implies $\forall I$, $Q_2(I) \subseteq Q_1(I)$

The Chase

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

- When $h : front(C) \to Q$,

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

$$C \vdash Q \cong step(C, Q)$$

- The *chase* is to *step* until a fixed point is reached.

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

Tableaux Minimization

- Given a query Q and set of EDs C
- we first chase Q with C to obtain U, a so-called *universal plan*
- then we search for sub-queries of U, chasing each in turn with C to check for equivalence with U.

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

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

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

$$
\begin{aligned}
\mathit{min}(\mathit{chase}(C, Q_1)) \;&=\; \texttt{for}\ (m_2\ \texttt{in}\ \mathit{Movies})\\
&\qquad \texttt{return}\ (m_2.\mathsf{director}, m_2.\mathsf{actor})
\end{aligned}
$$

Part 2

- Part 1:
  - Given a relational conjunctive query Q
  - and a set of constraints C of the form $\forall \vec{x}.\phi(\vec{x}) \rightarrow \exists \vec{y}.\psi(\vec{x}, \vec{y})$
  - we can compute a unique minimal query Q' such that $C \vdash Q \cong Q'$
  - or diverge
- Part 2:
  - Given a commutative, idempotent monad with zero in Coq
  - and a Coq monad comprehension Q
  - and a set of Coq proof objects C
  - our Coq tactic (semi) computes Q' and a proof that $C \vdash Q \cong Q'$

Coq

- Coq is a proof assistant based on functional programming with dependent types:

```
Inductive List (A : Type) : Nat → Type :=
 | nil : List A 0
 | cons : ∀(n : Nat), A → List A n → List A (n + 1).

Definition append A n m : List A n → List A m → List A (n + m)
    := ...
```

- Coq programs can be built interactively using a scripting language:

```
Theorem append_unit : ∀ A n m l, append A n m nil l = l.
Proof.
 intros; induction n;
   [ reflexivity | simpl in *; rewrite H; trivial ].
Qed.
```

- Coq is an intriguing ambient language for querying:

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

Queries in Coq

```
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).

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

Eval compute in (proj1 optimized_query).
(* = x ← Movies ; return (x.director, x.actor)
 *   : set (string × string)   *)
```

Idempotent, Commutative Monads

```
Class DataModel (M : Type → Type) : Type :=
{ Mret  : ∀ {T}, T → M T
; Mzero : ∀ {T}, M T
; Mbind : ∀ {T U}, M T → (T → M U) → M U
 (* plus many axioms, including
     for (x in X)(y in Y) = for (y in Y)(x in X)
     for (x in X)(x in X) = for (x in X)
  *)
}.
```

- Example: Finite sets
- `Mret v` $= \{v\}$
- `Mzero` $= \{\}$
- `Mbind m k` $= \bigcup_{x \in m} k(x)$. Write `x ← m ; k` for `Mbind m (fun x ⇒ k)`

Queries and EDs in Coq

```
(* 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)).
```

Tactic basics

- A tactic can examine this Coq code:

```
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).
```

- and normalize it into:

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

- and emit an equality proof using the monad laws.

Tactics, continued

- A Coq *proof goal* is a sequent, $\Gamma \vdash ? : t$, where Γ is a context of Coq terms and t is a Coq type.
- A tactic can transform a proof goal into new goals:

$$\Gamma \vdash ? : t \longrightarrow \{\Gamma' \vdash ?' : t', \ldots, \Gamma'' \vdash ?'' : t''\}$$

- or solve a proof goal by building a term from the context:

$$\Gamma \vdash ? : t \longrightarrow \Gamma \vdash e : t$$

- Our proof goals are queries and semantics-preservation proofs, and our transformations are re-write rules.

Tactics, continued

- Coq's tactics are designed for general-purpose theorem proving.
- So, the challenge is to map query optimization onto these tactics.
- This requires many structural lemmas, for example

$$(\forall x, Q(x) \cong Q'(x)) \longrightarrow \texttt{for}\ (x\ \texttt{in}\ X), Q(x) \cong \texttt{for}\ (x\ \texttt{in}\ X), Q'(x)$$

- and a tactic to exhaustively search for homomorphisms
- and tactics to match sub-terms of queries
- The payoff is a tactic that operates directly on Coq programs, rather than on a type of syntax for queries.

Analysis of the tactic-based approach

- Benefits:
  - Supports nested relations simply by proving new lemmas. (Contrast to deep-embedding approach)
  - Supports arbitrary Coq computation in `where` clauses with no effort.
  - Re-use of existing Coq infrastructure - higher-order unification, and backtracking search are built-in.
- Drawbacks:
  - Tactics are completely untyped, and so are error-prone to develop.
  - Many similar lemmas had to be proved.
  - Speed - finding homomorphisms is NP but $\mathcal{L}_{\text{tac}}$ is nonetheless slow.

Conclusion

- Part 1:
  - Given a relational conjunctive query Q
  - and a set of constraints C of the form $\forall \vec{x}.\phi(\vec{x}) \to \exists \vec{y}.\psi(\vec{x},\vec{y})$
  - we can compute a unique minimal query Q' such that $C \vdash Q \cong Q'$
  - or diverge
- Part 2:
  - Given a commutative, idempotent monad with zero in Coq
  - and a Coq monad comprehension Q
  - and a set of Coq proof objects C
  - our Coq tactic (semi) computes Q' and a proof that $C \vdash Q \cong Q'$
- Take-away:
  - Coq users can use our tactic to optimize monad comprehensions in a provably correct way.
  - Our work gives a *design pattern* for optimizing Coq code using tactics.
  - Toward dependently-typed LINQ!

Thanks to

- ONR grant N000141310260
- AFOSR grant FA9550-14-1-0031
- Lucian Popa