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
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 \(d\).

\[
\text{for } (m_1 \text{ in } Movies) \ (m_2 \text{ in } Movies) \\
\text{where } m_1.\text{title} = m_2.\text{title} \\
\text{return } (m_1.\text{director}, m_2.\text{actor})
\]

- Under functional dependency title \(\rightarrow\) director is equivalent to:

\[
\text{for } (m \text{ in } Movies) \\
\text{return } (m.\text{director}, m.\text{actor})
\]
Embedded Dependencies (EDs)

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

  \[
  \forall x \in X \quad \text{where} \quad P(x) \\
  \exists y \in Y \quad \text{where} \quad B(x, y)
  \]

- Functional dependency title $\rightarrow$ director expressed as:

  \[
  \forall (x \in Movies) (y \in Movies) \quad \text{where} \quad x.title = y.title, \\
  \exists \quad \text{where} \quad x.director = y.director
  \]
The front and back of an ED

\[ C := \forall (x \in X) \]
\[ \text{where } P(x) \]
\[ \exists (y \in Y) \]
\[ \text{where } B(x, y) \]

\[ \text{front}(C) := \forall (x \in X) \]
\[ \text{where } P(x) \]
\[ \text{return } x \]

\[ \text{back}(C) := \forall (x \in X) \exists (y \in Y) \]
\[ \text{where } P(x) \land B(x, y) \]
\[ \text{return } x \]

\[ \forall I, \ I \models C \iff \text{front}(C)(I) = \text{back}(C)(I) \]
Homomorphisms of queries

- A homomorphism $h : Q_1 \rightarrow Q_2$ between queries:

  \[
  \text{for } (v_1 \text{ in } V_1) \quad \text{for } (v_2 \text{ in } V_2)
  \]
  \[
  \text{where } P_1(v_1) \rightarrow_h \text{ where } P_2(v_2)
  \]
  \[
  \text{return } R_1(v_1) \quad \text{return } R_2(v_2)
  \]

- is a substitution $v_1 \mapsto v_2$ such that
  - $(h(v_1) \text{ in } V_1) \subseteq (v_2 \text{ in } V_2)$
  - $P_2(v_2) \vdash P_1(h(v_1))$
  - $P_2 \vdash R_1(h(v_1)) = R_2(v_2)$

- $Q_1 \rightarrow Q_2$ implies $\forall I, Q_2(I) \subseteq Q_1(I)$
The Chase

\[ C := \forall (x \in X) \quad Q := \forall (u \in V) \]

where \( P(x) \)

exists \((y \in Y)\)

where \( B(x, y) \)

\[ \text{When } h : \text{front}(C) \rightarrow Q, \]

\[ \text{step}(C, Q) := \forall (u \in V) (y \in Y) \]

where \( O(u) \land B(h(x), y) \)

return \( R(u) \)

\[ C \vdash Q \equiv \text{step}(C, Q) \]

\[ \text{The chase is to step until a fixed point is reached.} \]

\[ C \vdash Q_1 \equiv Q_2 \quad \text{if} \quad \text{chase}(C, Q_1) \leftrightarrow \text{chase}(C, Q_2) \]
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 $U$. 
\[ Q_1 \ := \ \text{for } (m_1 \text{ in } Movies) \ (m_2 \text{ in } Movies) \]
\[
\text{where } m_1.\text{title} = m_2.\text{title} \\
\text{return } (m_1.\text{director}, m_2.\text{actor})
\]

\[ C \ := \ \text{forall } (x \text{ in } Movies) \ (y \text{ in } Movies) \]
\[
\text{where } x.\text{title} = y.\text{title} \\
\text{exists} \\
\text{where } x.\text{director} = y.\text{director}
\]

\[ \text{chase}(C, Q_1) \ = \ \text{for } (m_1 \text{ in } Movies) \ (m_2 \text{ in } Movies) \]
\[
\text{where } m_1.\text{title} = m_2.\text{title} \land \\
m_1.\text{director} = m_2.\text{director} \\
\text{return } (m_1.\text{director}, m_2.\text{actor})
\]

\[ \text{min}(\text{chase}(C, Q_1)) \ = \ \text{for } (m_2 \text{ in } Movies) \\
\text{return } (m_2.\text{director}, m_2.\text{actor}) \]
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:

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

  ```coq
  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:

  ```coq
  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:

  ```coq
  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:
{qopt : set (string × string) | title_director_ed → qopt ≈ 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 \(x \leftarrow m ; k\) for Mbind m (fun x => k)
Queries and EDs in Coq

(* Queries *)

Definition query \{S T: Type\}
\[ (P : M S) (C : S \to \text{bool}) (E : S \to T) : M T := Mbind P (fun x \Rightarrow Mguard (C x) (Mret (E x))) \].

(* Embedded Dependencies *)

Definition embedded_dependency \{S S’: Type\}
\[ (F : M S) (Gf : S \to \text{bool}) (B : M S’) (Gb : S \to S’ \to \text{bool}) := Meq (query F Gf (fun x \Rightarrow x)) (query (Mprod F B) (fun ab \Rightarrow Gf (fst ab) && Gb (fst ab) (snd ab)) (fun x \Rightarrow fst x)). \]
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.
A Coq *proof goal* is a sequent, $\Gamma \vdash ? : t$, where $\Gamma$ 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

\[ (\forall x, Q(x) \equiv Q'(x)) \rightarrow \text{for (} x \text{ in } X, Q(x) \equiv \text{for (} x \text{ in } X, Q'(x) \]

- 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 $L_{\text{tac}}$ is nonetheless slow.
Conclusion

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

- 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