The Gap Between Theory & Practice

Most commercial relational query languages are based on the relational calculus. However in some respects they go beyond the formal model. They support:

- aggregate operators,
- sort orders,
- grouping,
- update capabilities.

New database languages must be able to handle:

- type extensibility,
- multiple collection types (e.g., sets, lists, trees, arrays),
- arbitrary nesting of type constructors,
- large objects (e.g., text, sound, image),
- methods.

A New Formal Model is Needed

A formal algebra:

- facilitates equational reasoning,
- provides a theory for proving query transformations correct,
- imposes language uniformity,
- avoids language inconsistencies.

What is an effective algebra?

Several aspects:

- coverage,
- ease of manipulation,
- ease of evaluation,
- uniformity.

Functional Languages to the Rescue

Functional languages are value-based (no side-effects!). Values are immutable; new values are constructed from old values. Programs are organized into functions.

Functional vs. imperative programs:

- Pure functions can always be tested separately.
- Many optimizations are not always valid in imperative languages:
  
  - $x + y \rightarrow y + x$
  
  - $x \times 0 \rightarrow 0$

  e.g., if $f() = \{a := a + 1; 5\}$, then $f() \times 0$ is not equivalent to $0$.

Modern Functional Languages

Most popular: SML and Haskell.

They are based on the lambda calculus.

They support:

- strong static typing with type inference,
- automatic garbage collection,
- resilience to store corruption (no core dumps!),
- parametric polymorphism,
- higher-order functions,
- algebraic data types (no pointers!),
- pattern matching.

Some features are showing up in new imperative languages (e.g., Java).

Example from Haskell

```haskell
data list a = Nil | Cons a (list a)

Cons 1 (Cons 2 (Cons 3 Nil))

Mapping a function $f$ over the elements of a list:

```
Loop Fusion and Deforestation

map f Nil = Nil
map f (Cons a r) = Cons (f a) (map f r)

sum Nil = 0
sum (Cons a r) = a + sum r

Suppose that we compose these operations:

\[ f(x) = \sum (\text{map} (\lambda a \rightarrow a+1) x) \]

A better definition of \( f \):

\[ f(Nil) = 0 \]
\[ f(\text{Cons} \ a \ r) = (a+1) + f(r) \]

How Can we Fuse Programs?

We can use standard program transformation techniques
... or we can use folds.

Folds

- can be defined for a large number of algebraic data types;
- support calculation-based program optimizations;
- support loop fusion and deforestation;
- facilitate equational reasoning and theorem proving.

So What is a fold?

The Fold Operator

A fold is the natural control structure for an algebraic data type. It uses functional parameters to abstract over common inductive patterns. It replaces data constructors with functions.

data list a = Nil | Cons a (list a)

\[ \text{fold} \ c \ n \ \text{Nil} = n \]
\[ \text{fold} \ c \ n \ (\text{Cons} \ a \ r) = c \ a \ (\text{fold} \ c \ n \ r) \]

For example, if \( x = \text{Cons} \ 1 \ (\text{Cons} \ 2 \ (\text{Cons} \ 3 \ \text{Nil})) \), then:

\[ \text{fold} \ c \ n \ x = c \ 1 \ (c \ 2 \ (c \ 3 \ n)) \]

Other Fold Operators

\[ \text{map} \ f \ Nil = Nil \]
\[ \text{map} \ f \ (\text{Cons} \ a \ r) = \text{Cons} \ (f \ a) \ (\text{map} \ f \ r) \]

Fusion Laws

For lists:

1. \( n' = g(n) \)
2. \( a \ (g \ r) = g(a \ r) \)
3. \( \text{fold} \ c \ n \ x = \text{fold} \ c' \ n' \ x \)

For trees:

1. \( n' = g(n) \)
2. \( (g \ l) \ a \ (g \ r) = g(a \ l \ r) \)
3. \( \text{fold} \ m \ n \ x = \text{fold} \ m' \ n' \ x \)
Fusion Example

map f x = fold (f a s -> Cons (f a) s) Nil x
= fold c n x

sum x = fold (a s -> a+s) 0 x

n' = sum n

= sum (fold c n x)
= fold c' n' x

n' = sum n = sum Nil = 0

= cons (f a) r
= sum (cons (f a) r)

=> c' a s = (f a)+s
=> sum (map f x) = fold (f a s -> (f a)+s) 0 x

Folds as a Basis for a Query Algebra

Folds have been used as a query algebra for an object-oriented database:

- relational database operations can be expressed as folds:
  join f p x y = fold (f a s -> if (p a b) then Cons (f a b) r else r) y r)

- query plans can be expressed as folds;
- the fusion algorithm generalizes many algebraic query optimization techniques (such as unnesting queries);
- fusion can be used for eliminating the object translation overhead when translating queries into plans.

Problems: set commutativity and idempotence:

x U y = y U x
x U x = x

Conclusion

Value-based, higher-order, operations:

- provide a uniform way of expressing database queries;
- have sufficient expressive power to capture modern database languages;
- satisfy simple laws that facilitate the proof of program correctness;
- support algebraic optimization methods.

Current Work

My current research work includes:

- building a query optimizer for ODMG'93 OQL using the monoid comprehension calculus. Using a physical design language to map conceptual queries into physical plans. (Work with Dave Maier at OGI; currently supported by NSF).
- extending OODB languages with temporal features (work with Ramez Elmasri at UTA).
- making functional languages more efficient by using program transformation techniques (work with Tim Sheard at OGI).